Implémentation efficace de l'arithmétique AMNS

Asma Chaouch

Université de Toulon
École doctorale 548 - Mer et Science
France

Ecole Nationale D'ingénieur de Monastir
Ecole doctorale - Sciences et Techniques pour l'Ingénieur
Univesité de Monastir
Tunisie

THÈSE présentée par :

Asma CHAOUCH

Pour obtenir le grade de

Docteur de l'Université de Toulon

et

Docteur de l'école Nationale D'ingénieur de Monastir

Spécialité : Génie électrique

Implémentation efficace de l'arithmétiques AMNS

Soutenue le 15/12/ 2021 devant le jury composé de :

M. DIDIER Laurent-Stéphane	Professeur, Université de Toulon, France	Directeur de thèse
M. BOURAOUI Ouni	Professeur, Université de Sousse, Tunisie	Co-Directeur de thèse
Mme MESNAGER Sihem	Professeur, Paris 8, France	Rapporteur
M. HAJJAJI Mohamed Ali	Maître de conférences HDR, Université de Sousse, Tunisie	Rapporteur
M. LIOUANE Noureddine	Professeur, Université de Monastir, Tunisie	Examinateur
M. DUQUESNE Sylvain	Professeur, Université de Rennes, France	Examinateur
Mme EL MRABET Nadia	Maître de conférence, EMSE Gardanne, France	Co-encadrant de thèse
M. BOUALLEGUE belgacem	Maître de conférence , Université de Monastir, Tunisie	Co-encadrant de thèse

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

AMNS Adapted Modular Number System

ECC Elliptic Curve Cryptography

VHSIC Very High Speed Integrated Circuits

VHDL VHSIC Hardware Description Language

Introduction et contexte de la thèse

Ces dernières années, la croissance continue de l'industrie a conduit à des estimations selon lesquelles d'ici 2025, il y aura 21,5 milliards d'appareils IoT activement connectés dans le monde [LTPK19]. L'augmentation massive de la transmission de données nécessite la confidentialité et l'intégrité des informations. Cela conduit à l'utilisation de primitives cryptographiques.

La cryptographie est le processus de conversion et de transfert de données dans un format inaccessible aux utilisateurs non autorisés. Elle est utilisée pour protéger différents types de données, telles que les données de carte de crédit, les données de courrier électronique et les données d'entreprise. Les objectifs fondamentaux de la cryptographie sont : d'abord le contrôle d'accès, puis l'authentification, puis la confidentialité, puis l'intégrité, et enfin la validité et la non-répudiation.

Dans la littérature, il existe deux types de cryptographie, qui sont la cryptographie à clé secrète et à la cryptographie clé publique [VSK20]. Le premier type, aussi appelé cryptographie symétrique, nécessite l'échange au préalable une clé qui sera utilisé pour les opérations de chiffrement et de déchiffrement. Comme exemple des schémas de chiffrement à clé secrète, on a le DES (Digital Encryption Standard) et l'AES (Advanced Encryption Standard). La cryptographie à clé publique, aussi appelé cryptographie asymétrique, ne nécessite pas cet échange de clé. Cependant, les schémas de ce type font intervenir des calculs plus complexes que ceux des schémas de chiffrement à clé secrète. En outre, il est indispensable de mettre en place un système d'authentification des clés publiques. On a par exemple comme standards de chiffrement à clé publique le cryptosystème RSA [RSA78a] et le cryptosystème ElGamal [ElG85].

La sécurité des cryptosystèmes à clé publique repose sur des problèmes mathématiques réputés difficiles. On a par exemple la factorisation de grandes entiers pour le RSA et la résolution du logarithme discret classique pour ElGamal. Avec les avancées dans ces domaines, ces schémas font depuis un certain nombre d'années place aux cryptosystèmes basés sur les courbes elliptiques

(ECC : Elliptic Curve Cryptography).

Ces cryptosystèmes reposent sur le problème de la résolution du logarithme discret sur les courbes elliptiques. C'est à dire trouver l'entier k à partir de deux points P et Q d'une courbe, tels que $Q = kP = P + P + \cdots + P$ (k fois). Jusqu'à présent, ce problème demeure de façon générale très difficile.

Comme conséquence, pour un niveau de sécurité équivalent, les cryptosystèmes basés sur les courbes elliptiques nécessitent des clés beaucoup plus petites que RSA ou ElGamal (classique).

Par exemple, pour un niveau de sécurité de 128 bits, une clé de taille 256 bits est nécessaire pour les schémas basés sur les courbes elliptiques là où le RSA nécessite une clé de taille 3072 bits [Mil86], [Kob87], comme le montre le tableau 1. Ce qui rend les schémas basés sur les courbes elliptiques particulièrement intéressants pour les environnements à puissance et mémoire réduites.

Niveau de sécurité	AES	RSA	ECC	HECC
	Taille de clé (bits)			
80	80	1024	160	60
112	112	2048	224	84
128	128	3072	256	96
192	192	7680	384	144
256	256	15360	512	192

TABLE 1 – Tableau comparatif de niveau de sécurité

L'implantation matérielle de protocole de chiffrement basé sur les courbes elliptiques nécessite plusieurs opérations sur plusieurs couches, comme présenté dans le tableau 2.

Opérations	couche
Protocoles	couche 4
Multiplication scalaire	couche 3
Addition et Doublement d'un point	couche 2
Arithmétique modulaire	couche 1

TABLE 2 – Classement des opérations sur l'ECC

Le tableau 2 présente 4 couches. Dans ce tableau, chaque niveau fait appel au niveau inférieur. Les opérations de la première couche sont : l'addition, la multiplication, le carré et l'inversion modulaires. L'addition et le doublement de points sont les opérations de la seconde couche. Le troisième niveau de ce

tableau correspond à l'opération principale de l'ECC. Il s'agit de la multiplication scalaire (ECSM : Elliptic Curve Scalar Multiplication). Finalement, les opérations de la couche 4 sont des protocoles tels que l'ECDH (Elliptic Curve Diffie Hellman) et l'EC ElGamal.

L'efficacité des protocoles est un critère essentiel pour leurs utilisation en pratique. L'optimisation de ces protocoles peut être faite sur une ou plusieurs des couches présentées dans le tableau 2. Dans cette thèse, nous nous sommes intéressés à l'utilisation des solutions arithmétiques pour les couches 1, 2 et 3. Nous abordons dans un premier temps la couche 1, pour calculer la multiplication modulaire en particulier. Nous nous penchons ensuite sur les couches 2 et 3.

Un autre critère fondamental à l'utilisation des protocoles cryptographiques est leurs résistances aux attaques par canaux auxiliaires. En effet, ces attaques ont été prouvées efficaces dans bon nombre de scénarios. On a par exemple les attaques proposées par Coron [Cor99]. La protection de ces protocoles contre ces attaques passe par l'ajout de contre-mesures dans une ou plusieurs des couches du tableau 2, selon l'attaque dont on veut se prémunir. Dans cette thèse, nous nous sommes penchés sur la randomisation de l'arithmétique modulaire (donc la couche 1), qui est une contre-mesure indispensable contre les attaques de type DPA (Differential Power Analysis) [KJJ99].

Pour implémenter ces architectures matérielles, plusieurs technologies peuvent être utilisées, dont les circuits intégrés spécifiques aux applications (ASIC) sont les plus performants. Cependant, le processus de fabrication de ce matériel est très coûteux. Ce coût ne peut être compensé que par une production élevée. La meilleure solution consiste à utiliser des circuits reconfigurables après la fabrication, comme les réseaux de portes programmables (FPGA : Field Programmable Gate Array). La flexibilité des solutions sécurisées peut être obtenue grâce aux FPGA, qui sont récemment devenus de plus en plus populaires. Les FPGA sont reprogrammables et tous les LUT restants peuvent encore être utilisés pour des fonctionnalités supplémentaires. En particulier, l'utilisation de l'IP (propriété intellectuelle) facilite également la reconfiguration du FPGA en le réalisant des dizaines de fois sur le FPGA et en effectuant de nombreuses opérations similaires simultanément.

Cette thèse présente plusieurs contributions à cette évolution, qui sont toutes liées aux calculs arithmétiques.

Objectifs et orientation de la thèse

Dans nos recherches, nous contribuons à concevoir des nouvelles méthodologies et architectures d'implémentation de la multiplication modulaire ainsi que la multiplication scalaire sur les courbes elliptiques, de manière efficace et sécurisée. Plus précisément, nous concevons des architectures basées sur le système de représentations modulaire adapté (AMNS : Adapted Modulaire Number System), proposé par Bajard et al. [BIP04] en 2004. Il s'agit d'un système de numération non positionnel pour représenter les entiers modulo un entier donné. Dans ce système, les éléments sont représentés sous forme de polynômes avec de petits coefficients. Aucune implémentation matérielle de ce système n'a été faite auparavant ; nos travaux sont donc un premier pas dans ce sens.

Nous avons conçu une nouvelle architecture optimisée pour la multiplication scalaire dans ce système. L'objectif étant d'intégrer cette architecture à une plus complexe destinée à effectuer la multiplication scalaire sur les courbes elliptiques. Nous proposons également une architecture pour la multiplication modulaire avec la randomisation proposée dans [DDEM+19a]. Nous verrons que les architectures proposées offrent flexibilité et rapidité.

Contribution de la thèse

Dans cette thèse, nous proposons des nouvelles architectures pour effectuer la multiplication modulaire des grands entiers. Ces dernières seront par la suite intégrées à une architecture pour la multiplication scalaires sur courbes elliptiques. Nos principales contributions peuvent être résumées comme suit :

— Le chapitre 2 présente la première contribution, qui est une nouvelle implantation faite pour la multiplication modulaire (MM) dans les systèmes de représentation AMNS. Pour la multiplication scalaire ECC, cette opération est très coûteuse. Notre proposition est d'adapter l'algorithme de multiplication modulaire en AMNS en utilisant les directives de l'outil de synthèse haut niveau HLS. Ces résultats ont été présentés dans une conférence internationale [CDD+19].

— La deuxième partie est introduite au chapitre 3. Il s'agit de la conception d'une architecture plus rapide pour implémenter une multiplication modulaire en AMNS. En effet, nous avons conçu deux architectures matérielles décrites en VHDL. La première est une architecture séquentielle et la deuxième est une architecture semi-pipeline. Les résultats obtenus montrent que l'architecture semi-pipeline est la plus rapide par rapport à

l'état de l'art. Ces résultats ont été publiés dans la revue JISA (Journal of Information Security and Applications) [CDD+21].

— Au chapitre 4, nous avons introduit la troisième contribution. Il s'agit de sécuriser l'implantation matérielle d'une MM en AMNS contre les attaques de type DPA. Nous avons conçu une architecture matérielle semi-pipeline pour l'algorithme de randomisation de la MM en AMNS. Les résultats obtenus montrent que ces architectures sont performantes.

— Enfin, le chapitre 5 introduit la quatrième contribution. Il s'agit d'intégrer l'accélérateur matériel semi-pipeline de la MM dans une architecture matérielle pour la multiplication scalaire sur l'ECC (ECSM).

Organisation de la thèse

Les chapitres de cette thèse sont organisés comme suit :

Dans le chapitre 1, nous présentons un état de l'art sur toutes les notions de base sur lesquelles se fondent les autres chapitres. Cela comporte une étude bibliographique de l'arithmétique modulaire, du système de représentation modulaire adapté (AMNS), ainsi qu'une étude des techniques d'implantation matérielle d'un cryptoprocesseur sur FPGA.

Dans le chapitre 2, nous concevons une IP (Intellectual Proprety) matérielle pour l'implémenter sur FPGA. Notre méthode est basée sur la synthèse de haut niveau (HLS), un outil qui peut être utilisé pour générer des implantations FPGA à partir de descriptions basées sur des langages de programmation de haut niveau tels que le C ou le C ++. Nous introduisons d'abord le FPGA et le HLS. Ensuite, nous utilisons les directives pour améliorer l'algorithme de multiplication modulaire afin d'obtenir une implantation efficace. Finalement, nous implémentons la multiplication modulaire sur les FPGAs en utilisant Vivado HLS. De plus, nous utilisons "Vivado Design Suite" pour interfacer notre IP avec les ressources de stockage et pour générer une description HDL standard et la mapper à la partie FPGA. Finalement, nous comparons nos résultats d'implantation avec l'état de l'art.

Le chapitre 3 propose deux nouvelles architectures matérielles pour implémenter la multiplication modulaire sur le corps finis. Nous verrons que ces architectures sont plus rapides que celle présenté dans le chapitre 2 ainsi que celles proposées dans la littérature. La première architecture est séquentielle et la seconde, plus rapide que la première, est semi-parallèle. Finalement, nous implémentons ces architectures pour des modules de tailles $128, 256, 512$ et 1024 bits et avec des mots de tailles $w = 18, 36, 64$ bits.

Dans le chapitre 4, nous proposons une nouvelle architecture, qui est une adaptation de l'architecture présentée dans le chapitre 3, pour introduire la randomisation dans la multiplication modulaire selon l'approche proposée dans [DDEM+19a]. Ensuite, nous analysons le résultat de l'implantation matérielle pour un module de 256 bits.

Le chapitre 5 porte sur notre architecture matérielle pour effectuer la multiplication scalaire sur les courbes elliptiques. Nous commençons ce chapitre par une introduction sur les courbes elliptique ainsi que les techniques de multiplication scalaire. Ensuite, nous concevons l'architecture matérielle pour la multiplication scalaire en nous basant sur l'architecture pour l'arithmétique modulaire dans le système AMNS, proposée dans le chapitre 3. Enfin, nous analysons les résultats d'implantation sur FPGA pour des modules de 256 bits.

1 État de L'art

Sommaire

Cette thèse porte sur l'amélioration de l'implantation matérielle de l'arithmétique modulaire cryptographiques asymétrique. Plus précisément, nous nous focalisons sur les architectures permettant d'accélérer l'arithmétique modulaire dans les systèmes de représentation modulaire adapté, appelés AMNS (Adapted Modular Number System) en anglais. En effet, l'efficacité du calcul arithmétique modulaire des grands entiers est primordiale pour la cryptographie asymétrique.

Dans ce chapitre, nous allons présenter les notions qui servent le propos de cette thèse. Nous donnons dans la partie 1.1 une brève introduction au

fondement mathématique de l'arithmétique modulaire dans un corps fini, une définition formelle des AMNS est donnée, introduit par Bajard et al. en 2004 [BIP04]. La deuxième partie 1.2 est consacrée à la présentation des différentes méthodes d'implantation de la multiplication modulaire sur de grands entiers.

1.1 Cryptographie et arithmétique modulaire

Un cryptosystème ou «système cryptographique» est un ensemble d'algorithmes qui permet principalement le chiffrement et le déchiffrement des messages que les utilisateurs souhaitent envoyer à quelqu'un en secret. Généralement, un tel mécanisme est composé de 3 étapes différentes : génération de clés, chiffrement des messages et déchiffrement des messages.

Pour ce faire, la cryptographie asymétrique utilise des opérations sur des corps fini, telles que la multiplication, l'addition, la soustraction, la division, l'inversion. Ces opérations nécessitent une réduction modulaire. D'autre part, la multiplication modulaire est l'opération la plus importante en cryptographie.

Dans la partie qui suit, nous introduisons la cryptographie asymétrique. Après les avoir présentées, nous citons les méthodes de réductions modulaires ainsi que les algorithmes de multiplication modulaire. Ensuite, nous comparons les résultats d'implantation matérielle dans le système de représentation AMNS avec les algorithmes existants.

1.1.1 Chiffrement à clé publique

Diffie et Hellman ont proposé pour la première fois le concept de cryptographie asymétrique en 1975 [DH76], appelé aussi cryptographie à clé publique. Ce nom vient du fait qu'une des clés utilisées doit être accessible publiquement : tout le monde peut utiliser la clé publique et chiffrer des messages, mais seul le détenteur de la clé sécrète, associée à cette clé publique, peut déchiffrer ces messages.

Depuis, divers algorithmes à clé publique ont été proposés. Les algorithmes PKC (Public Key Cryptography) les plus utilisés aujourd'hui sont RSA [RSA78b] et les protocoles basés sur les courbes elliptiques ou hyperelliptiques [Kob89]. La mécanisme de chiffrement/déchiffrement est décrit de la façon suivante :

$$C = E(K_{pub}, M) \tag{1.1}$$

$$M = D(K_{piv}, C) \tag{1.2}$$

avec,
E : la fonction de chiffrement
D : la fonction de déchiffrement
M : le message à chiffré
C : le message chiffré
K_{pub} : la clé publique
K_{piv} : la clé privée

Ces fonctions sont dites unidirectionnelles. Ils constituent le principe fondamental des algorithmes à clé publique. D'une part, ils sont assez simples à calculer pour chaque entrée, d'autre part ils sont très complexes à inverser. Mathématiquement parlant, le calcul de $y = f(x)$ devrait être suffisamment rapide, alors que le calcul inverse $x = f^{-1}(y)$ devrait être d'une lenteur inacceptable, selon [PP09]. Comme exemples de telles fonctions, on peut citer :

— **La factorisation de grands entiers**

Par exemple, le cryptosystème RSA (Rivest Shamir Adleman) [BB79] utilise la difficulté de factorisation entière.

— **Le calcul de logarithme discret classique**

Comme exemples de cryptosystèmes reposant sur la difficulté du logarithme discret classique, on peut citer ElGamal et DSA (Digital Signature Algorithm [RSA78b], utilisé pour la signature numérique).

— **Le calcul du logarithme discret sur courbes elliptiques**

Ici, on peut citer le protocole ECDSA (Elliptic Curve Digital Signature Algorithm) [JMV01] qui est une évolution du DSA basée sur les courbes elliptiques.

1.1.2 Histoire de l'arithmétique modulaire

L'arithmétique modulaire sert de brique de base à bon nombre de cryptosystèmes à clé publique. Les opérations principales de l'arithmétique modulaire sont l'addition et la multiplication modulaires. Pour effectuer efficacement ces opérations, il est indispensable que la réduction modulaire soit efficace.

Dans cette partie, nous abordons les techniques de réduction les plus couramment utilisées afin de les exploiter dans les mises en oeuvre matérielles présentées dans les chapitres 2 et 3.

1.1.2.1 Principe de la réduction modulaire

La réduction modulaire correspond au calcul du reste de la division euclidienne. Mathématiquement, elle s'écrit $r = a \bmod q$. La division étant de façon générale très coûteuse, plusieurs méthodes ont été proposées pour effectuer la réduction modulaire sans avoir à effectuer la division non triviale.

Parmi les algorithmes de réduction modulaire les plus connus, il y a la réduction de Barrett [Bar86] et la réduction de Montogomery [Mon85].

Dans cette partie, nous allons présenté les algorithmes classique de la réduction de Barrett, de Montgomery, Pseudo Mersenne et Mersenne généralisé.

A) **Réduction modulaire de Barrett** L'algorithme de réduction de Barrett a été introduit par P.D. Barrett [Bar86] pour optimiser l'opération de réduction modulaire en remplaçant les divisions par des multiplications, afin d'éviter les divisions coûteuses. Le principe de base de la réduction de Barrett est d'utiliser la division d'un premier q donné pour pré-calculer un facteur. Cela implique de n'utiliser que des opérations de multiplication, de soustraction et de décalage.

Algorithm 1 Algorithme de réduction de Barrett

Entrées: $a \in \mathbb{Z}_{\prime\prime}/f(x)$ avec q un nombre premier, $k = log_2(q)$ avec $k \in \mathbb{N}$ tels que $2^k > q$, $r = \left\lfloor \frac{4^k}{q} \right\rfloor$.

Sorties: $t = a - \left\lfloor \frac{ar}{4^k} \right\rfloor q$

1: **si** $t < q$ **alors**
2: return t $\#t = a \bmod q$
3: **sinon**
4: return $t - q$ $\#t = a \bmod q$
5: **fin si**

Ces opérations sont plus rapides que l'opération de la division. Les étapes de l'algorithme de réduction de Barrett sont présentées dans l'algorithme 1, et sa méthode est la suivante. Puisque le module q est connu à l'avance et que le facteur r ne dépend que de ce module, il peut être pré calculé et stocké. Ensuite, la fonction de réduction n'a besoin que de calculer la valeur restante t. Lors du calcul de la valeur de t, nous pouvons utiliser l'opération de décalage vers la droite pour diviser 4^k par une puissance de 2. Par conséquent, l'ensemble du calcul est réduit à deux multiplications, à savoir l'opération de décalage à droite et l'opération de soustraction.

B) **Réduction de Montgomery**

Les différentes opérations de l'algorithme de réduction de Montgomery [Mon85] se trouvent dans l'agorithme 2. Cet algorithme calcule la réduction de $(X\ M^{-1} \bmod p)$, au lieu de la réduction de $(X \bmod p)$. Ici M est premier avec p et l'inverse de M module p existe.

Algorithm 2 Algorithme de réduction de Montgomery

Entrées: p, M premier avec p, X et $\beta = -p^{-1} \bmod M$

Sorties: $XM^{-1} \bmod p$

1: $Q \leftarrow (X \times \beta) \bmod M$
2: $R = X + p \times Q$
3: $S \leftarrow R/M$
4: return S

L'idée dans cet algorithme est de chercher la valeur positive la plus petite de Q, telle que $R = X + Q \times p \equiv X \pmod p \equiv 0 \pmod M$. Pour cela, la valeur de $\beta = -p^{-1} \bmod M$ doit être calculée à l'avance, de sorte que $Q = X \times \beta \bmod M$.

C) **Réduction modulaire par Pseudo Mersenne**

En 1644, Marin Mersenne a émis l'hypothèse que les nombres de la forme $p = 2^k - 1$ sont des nombres premiers avec des entiers $k \in \mathbb{N}$. En 1992, Richard Crandall [Cra92] a étendu les nombres de Mersenne et proposa les pseudo Mersenne, qui sont définits par le module p de la forme $p = 2^k - c$, où c est un petit entier. Il est également très efficace d'utiliser des nombres pseudo Merssene pour la réduction modulaire [LLMP93].

Algorithm 3 Algorithme de réduction modulaire pour un module pseudo Mersenne

Entrées: s : Un module $m = 2^r - c, r \in \mathbb{N}^*, c \in [0, \sqrt{m}[$, et un entier $a \in [0, 2^{2r}[$

Sorties: $|a|_m \in [0, m[$

1: $(a_1, a_2) \leftarrow (a >> r, a \ \ and \ \ (2^r - 1))$
2: $b \leftarrow a_1 \times c$
3: $(b_1, b_2) \leftarrow (b >> r, b \ \ and \ \ (2^r - 1))$
4: $d \leftarrow b_1 \times c$
5: $r \leftarrow (a_2 + b_2 + d) \ \bmod m$

D) **Réduction modulaire avec les nombres de Mersenne généralisés**
En 1999, Jérôme Solinas a proposé le Nombre de Mersenne Généralisé (GMN). GMN est exprimé sous la forme d'un polynôme de poids faible comme $p = f(t)$, où t est une puissance de 2, et le coefficient du polynôme du poids le plus faible $f(t)$ est très petit par rapport à t. Si le module est GMN, alors la réduction modulaire ne nécessite qu'une simple addition, multiplication et soustraction d'entiers. Toutefois, cette méthode présente deux inconvénients. Le premier inconvénient est qu'il n'existe pas beaucoup de GMN utiles, le deuxième inconvénient est qu'il faut créer une architecture matérielle spécifique pour chaque GMN [Cur99].

1.1.2.2 Méthodes de multiplication modulaire

Les algorithmes que nous présentons dans cette partie correspondent aux algorithmes de multiplication modulaire utilisés dans l'étude comparative des résultats d'implantations dans les chapitres suivants. Dans cette partie, nous mettons en évidence les deux méthodes suivantes : CIOS (Coarsely Integrated Operand Scanning) et la multiplication modulaire en RNS (Residue Number Systems, en anglais). La méthode CIOS entrelace les opérations de multiplication et de réduction, alors que le RNS effectue ces deux opérations séparément.

A) **Multiplication de Montgomery CIOS**
L'algorithme de multiplication modulaire de Montgomery est proposé dans [Mon85]. Cette méthode remplace la division euclidienne par deux multiplication et une division triviale (par puissance de deux). Cela implique une faible consommation de ressources. Par la suite, il fournit un algorithme rapide pour calculer la multiplication et la réduction modulaires.

Cette méthode coûte $3n^2$ multiplications pour une opérandes de n bits. Dans [DK91], Dussé et Kaliski améliorent cette multiplication modulaire en réduisant le nombre de petites multiplications à $2n^2 + n$.

L'algorithme CIOS coûte $2n^2 + n$ multiplications, en plus de $4n^2 + 4n + 2$ opérations d'écriture et de lecture. Pour que la méthode de Montgomery fonctionne efficacement, le module doit être impair afin que la division par une seule puissance de deux soit possible dans l'étape de réduction.

Nous supposons que $c = (c[0], \ldots, c[n-1])$ dans la représentation *little-endian* (c'est-à-dire que la valeur la moins significative dans le n-uplet est stockée en premier), avec $c[0]$ est un modulo inversible $w = 2^k$ où k est la taille de l'architecture considérée. Par conséquent, l'entier $g = -c[0]^{-1}$

Algorithm 4 Montgomery CIOS [KAK96]

Entrées: $a \in \mathbb{Z}/p\mathbb{Z}$ et $b \in \mathbb{Z}/p\mathbb{Z}$

Sorties: $t = ab \pmod p$

1: $t \leftarrow (0, \ldots, 0)$ #Il faut initialiser le tableau t à 0.
2: **pour** $i = 0 \ldots n-1$ **faire**
3: $\quad C \leftarrow 0$
4: $\quad$ **pour** $j = 0 \ldots n-1$ **faire**
5: $\quad\quad (C, S) \leftarrow t[j] + a[j] * b[i] + C$
6: $\quad\quad t[j] \leftarrow S$
7: $\quad$ **fin pour**
8: $\quad (C, S) \leftarrow t[n] + C$
9: $\quad t[n] \leftarrow S$
10: $\quad t[n+1] \leftarrow C$
11: $\quad m \leftarrow g * t[0] \pmod W$
12: $\quad (C, S) \leftarrow t[0] + m * p[0]$
13: $\quad$ **pour** $j = 1 \ldots n-1$ **faire**
14: $\quad\quad (C, S) \leftarrow t[j] + m * p[j] + C$
15: $\quad\quad t[j-1] \leftarrow S$
16: $\quad$ **fin pour**
17: $\quad (C, S) \leftarrow t[n] + C$
18: $\quad t[n-1] \leftarrow S$
19: $\quad t[n] \leftarrow t[n+1] + C$
20: **fin pour**
21: return t # $(t[0], \ldots, t[n])$.

(mod w) existe. Cet entier est utilisé pour toutes les opérations présentées dans [KAK96]. L'algorithme 4 représente la multiplication modulaire CIOS. Pour cet algorithme, nous supposons que les entiers sont représentés sous forme de tableaux de taille n où chaque colonne est de taille w bits. C-à-d si $c \in \mathbb{Z}/p\mathbb{Z}$, alors $c = (c[0], \ldots, c[n-1])$.

L'algorithme 4 utilise un tableau tab de taille $n+2$. Elle fournit en sortie le premier élément $n+1$ de tab ; c-à-d $(tab[0], \ldots, tab[n])$.

B) **La multiplication modulaire de Montgomery dans le RNS**

Le système de représentation modulaire RNS (Residue Number Systems) est une autre façon de représenter le nombre modulaire. Ces entiers

peuvent être représentés par un ensemble limité de petites valeurs indépendantes, ainsi les opérations arithmétiques de base peuvent être effectuées directement sur ceux-ci en parallèle sans aucun mécanisme de propagation de retenue.

L'algorithme 5 est l'algorithme de Montgomery adapté aux systèmes de représentation RNS présenté dans [BDK98], calculant $R \equiv ABM^{-1} \bmod N$.

Algorithm 5 Algorithme de multiplication modulaire de Montgomery en RNS

Entrées: $M = \prod_{i=1}^{n} m_i$, $A = \sum_{i=1}^{n} a'_i M$, $B \in RNS$, $GCD(M, N) = 1$

Sorties: $R \equiv ABM^{-1} \bmod N$

1: **pour** $i = 1 \dots n$ **faire**
2: $\quad q'_i \leftarrow (r_i + a'_i \times b_i) \times (m_i - n_i)_i^{-1} \bmod m_i$
3: $\quad R \leftarrow R + (a_i \times B) + (q'_i \times N)$
4: $\quad R \leftarrow \frac{R}{m_i}$
5: **fin pour**

C) **Multiplication modulaire avec les nombres de Mersenne**

En 1981, Knuth [Knu81] a proposé l'algorithme de multiplication modulaire avec les nombres de Mersenne que nous présentons ici. Un nombre de Mersenne p a une forme particulière de la forme $p = 2^k - 1$ avec $k \in \mathbb{N}$. Cette forme implique l'équivalent de $2^k \equiv 1 \pmod{p}$. Cette caractéristique permet de simplifier énormément la réduction, en la remplaçant par une addition.

Algorithm 6 Algorithme de multiplication modulaire sur les nombres de Mersenne

Entrées: a, b avec $0 \leqslant a, b < p$, $p = 2^k - 1$

Sorties: $c = ab \bmod p$

1: $r \leftarrow ab$
2: $r = r_1 2^k + r_0$
3: $c \leftarrow r_1 + r_0$
4: **si** $c \geqslant p$ **alors**
5: $\quad c \leftarrow c - p$
6: **fin si**

1.1.3 Système de représentation modulaire adapté

Le système de représentation modulaire adapté (AMNS : Adapted Modular Number System, en anglais) a été introduit par Thomas Plantard [Pla05] en 2005, afin d'accélérer l'arithmétique modulaire. Les opérations de l'arithmétique modulaire, telles que l'addition, la multiplication et la division, sont le cœur de calcul des algorithmes de chiffrement à clé publique, tels que RSA et ElGamal. La principale caractéristique de l'AMNS est que les éléments y sont présentés sous forme de polynômes.

Grâce à cette caractéristique, les algorithmes pour effectuer les opérations arithmétiques dans ce système sont sans branchement conditionnel et offrent des possibilités de parallélisation.

Dans [Dos20], Yssouf Dosso montre dans sa thèse comment générer des AMNS pour un entier premier quelconque, afin d'effectuer des opérations modulaires de manière efficace. Les résultats d'implémentation logicielle présentés dans cet article montrent que l'AMNS est plus efficace que les bibliothèques très connues telles que openSSL et GNU-MP, pour l'arithmétique modulaire entre 256 bits et 1024 bits. Dans cette partie, nous donnons un aperçu de ce système de représentation et nous présentons ses algorithmes de base.

Définition 1.1. Un système de représentation modulaire adapté (AMNS) est défini par un tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$ tel que pour chaque entier $0 \leqslant x < p$, il existe un vecteur $V = (v_0, \ldots, v_{n-1})$ tels que :

$$x = \sum_{i=0}^{n-1} v_i \gamma^i \bmod p \ ,$$

Avec :

- $|v_i| < \rho, \quad \rho \approx p^{1/n}$
- γ est une racine modulo p du polynôme $E(X)$,
- le polynôme $E(X) = X^n - \lambda$ est nommé le polynôme de réduction externe.
- λ un petit entier non nul ($\lambda = \pm 1, \pm 2$ ou ± 3).

Dans ce cas, on dit que le polynôme $V(X) = v_0 + v_1 X + \cdots + v_{n-1} X^{n-1}$ est une représentation de x dans $\mathcal{B}$ et on note $V \equiv x_{\mathcal{B}}$.

Les opérations arithmétiques principales de l'AMNS sont l'addition et la multiplication. Les éléments dans ce système sont des polynômes. De ce fait,

l'addition et la multiplication dans ce système de représentation s'appuie sur une simple addition et multiplication polynomiale. Cependant, il est nécessaire d'effectuer des opérations supplémentaires pour obtenir le résultat de ces opérations en AMNS. Nous expliquons pourquoi dans la section suivante.

1.1.3.1 Arithmétique modulaire en AMNS

Soient $x, y \in \mathbb{Z}/p\mathbb{Z}$ deux entiers. Soient $V \equiv x_{\mathcal{B}}$ et $W \equiv y_{\mathcal{B}}$ leurs représentations dans $\mathcal{B}$. Pour que l'addition polynomiale $S = V + W$ soit une représentation de $x + y$ dans $\mathcal{B}$, une opération appelée *réduction interne* doit être appliquée. Il en est de même pour la soustraction. En effet, même si $S(\gamma) \equiv (x + y) \pmod{p}$ et $\deg(S) < n$, les coefficients pourraient être supérieurs ou égaux en valeurs absolues à ρ. La *réduction interne* est l'opération qui réduit la taille des coefficients polynomiaux, cette primitive sera décrite dans la partie B).

Finalement, avant de présenter les algorithmes de réduction, nous devons établir quelques notations et conventions pour simplifier les équations :
Par souci de cohérence, nous supposerons que $p \geqslant 3$ et $n, \gamma, \rho \geqslant 1$.

$\mathbb{Z}_n[X]$ représente l'ensemble des polynômes de $\mathbb{Z}[X]$ ayant un degré inférieur à n :

$$\mathbb{Z}_n[X] = \{C \in \mathbb{Z}[X], \text{ tel que : } \deg(C) < n\}.$$

Avec $C \in \mathbb{Z}[X]$ un polynôme, $C \bmod (E, \phi)$ représente la réduction du polynôme $C \bmod E$ dont les coefficients ont été réduits modulo ϕ.

1.1.3.2 Multiplication modulaire en AMNS

Comme indiqué ci-dessus, Negre et Plantard dans [NP08] proposent une méthode de multiplication modulaire qui inclut une méthode de réduction de type Montgomery. L'algorithme 7 décrit leur proposition, à la différence qu'il a été décomposé en plusieurs étapes ici.

Dans un AMNS, la multiplication polynomiale de $C = A \times B$ satisfait $C(\gamma) \equiv xy \pmod{p}$, avec $A \equiv x_{\mathcal{B}}$ et $B \equiv y_{\mathcal{B}}$ des représentations de x, y dans $\mathcal{B}$, où $x, y \in \mathbb{Z}/p\mathbb{Z}$ deux entiers.

Cependant, nous ne pouvons pas écrire que $C \equiv (xy)_{\mathcal{B}}$. En effet, pour que T soit une représentation de xy dans $\mathcal{B}$, il faut que le degré de C soit inférieur strictement à n et il faut que les coefficients de C soient strictement inférieure en valeur absolue à ρ. Pour cela, deux primitives doivent être appliquées : la réduction externe et la réduction interne. La *réduction externe* est l'opération

Algorithm 7 Multiplication modulaire en AMNS [BIP05]

Entrées: $A \in \mathcal{B}$, $B \in \mathcal{B}$ and $\mathcal{B} = (p, n, \gamma, \rho, E)$

Sorties: $S \in \mathcal{B}$ with $S(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod{p}$

1: $C \leftarrow A \times B$
2: $C' \leftarrow \text{RedExt}(C)$
3: $S \leftarrow \text{RedCoeff}(C')$
4: return S

qui permet de maintenir le degré de T inférieur à n. Cela se fait avec la réduction de polynôme modulo E sur le produit VW, cette primitive sera détaillée dans la partie A). La *réduction interne* est l'opération qui réduit la taille des coefficients polynomiaux, cette primitive sera décrite dans la partie B).

Dans [NP08], Negre et Plantard ont montré que si les paramètres ϕ et ρ sont tels que

$$\rho \geqslant 2n|\lambda|\|M\|_\infty \quad \text{et} \quad \phi \geqslant 2n|\lambda|\rho$$

alors, la valeur de la sortie S de l'algorithme 7 est telle que $\|S\|_\infty < \rho$ (i.e. $S \in \mathcal{B}$). Cette exigence est supposée dans l'algorithme 7.
Pour tout nombre entier premier $p \geqslant 3$, Didier et al. [DDV19] prouvent qu'il est toujours possible de générer un AMNS, dont le paramètre E est tel que $E(X) = X^n - \lambda$, avec $\lambda = \pm 2^i$, où $i \in \mathbb{N}$.

Les réductions interne et externe sont ainsi plus rapides, vu que ces opérations impliquent de nombreuses multiplications par λ. Comme mentionné plus haut, ϕ doit être une puissance de deux, autrement dit $\phi = 2^t$, avec $t \in \mathbb{N} \setminus \{0\}$.

A) **La réduction externe** est une réduction polynomiale, dont l'objectif est de maintenir le degré des représentations des AMNS inférieur à n. L'algorithme 8 de réduction externe est proposé par Plantard [Pla05] (Algorithme 28). La réduction externe du polynôme C est l'opération $R = C \ mod \ E$, où E est le polynôme de réduction externe de la forme $E(X) = X^n - \lambda$, avec λ très petit (elle peut avoir une valeur entre $-10, -9, \ \ldots, \ et 10$).

B) **La réduction interne** a pour objectif de garantir que la valeur absolue des coefficients des polynômes est inférieure à ρ.

Dans le cas de la multiplication modulaire dans AMNS, la réduction interne intervient après la réduction externe.

Algorithm 8 RedExt (Réduction externe) [Pla05]

Entrées: $C \in \mathbb{Z}[X]$ avec $\deg(C) < 2n - 1$ et $E(X) = X^n - \lambda$

Sorties: vous que $R \in \mathbb{Z}_n[X]$, de sorte que $R = C \bmod E$

1: **pour** $i = 0 \dots n - 2$ **faire**
2: $\quad r_i \leftarrow c_i + \lambda c_{n+i}$
3: **fin pour**
4: $r_{n-1} \leftarrow c_{n-1}$
5: return R # $R = (r_0, \quad dots, \ r_{n-1})$

Dans [BIP04, BIP05, NP08], d'autres méthodes de réduction interne ont été présentées. Dans les cas où la valeur du module p n'est pas imposé, mais uniquement sa taille approximative, la méthode proposée dans [BIP04] est la plus efficace. Cependant, lorsque la valeur de p est déjà fixée, alors la proposition de [NP08] est actuellement la meilleure, grâce à son coût théorique et ses besoins en mémoire. C'est sur cette méthode que nous nous focalisons dans cette thèse.

Dans [NP08], Negre et Plantard ont proposé une approche de réduction permettant d'effectuer la réduction interne. En Combinant la réduction interne de Montgomery avec l'opération de multiplication. Dans le cadre de cette thèse, nous mettons à part le processus de multiplication de la réduction interne, puisque cette réduction est également essentielle pour d'autres opérations comme l'addition et la soustraction. L'algorithme 9 est celui de la réduction interne qui est présenté dans [NP08].

Algorithm 9 RedCoeff (réduction des coefficients) [NP08]

Entrées: $\mathcal{B} = (p, n, \gamma, \rho, E)$, $V \in \mathbb{Z}_n[X]$, $M \in \mathcal{B}$ tel que $M(\gamma) \equiv 0 \pmod p$, $\phi \in \mathbb{N} \setminus \{0\}$ et $M' = -M^{-1} \bmod (E, \phi)$.

Sorties: $S(\gamma) = C'(\gamma)\phi^{-1} \pmod p$, avec $S \in \mathbb{Z}_n[X]$

1: $Q \leftarrow C' \times M' \bmod (E, \phi)$
2: $R \leftarrow (C' + (Q \times M) \bmod E)$
3: $S \leftarrow R/\phi$
4: return S

On constate que cette méthode de réduction de Montgomery ajoute trois paramètres supplémentaires : deux polynômes M et M' et un entier non nul ϕ. Il est important de noter que ces polynômes sont tels que $M \in \mathcal{B}$, $M(\gamma) \equiv 0 \pmod{p}$ et $M' = -M^{-1} \bmod (E, \phi)$. Pour que cet algorithme soit efficace, il est important de prendre ϕ comme une puissance de deux, car cet algorithme fait intervenir plusieurs réductions modulo ϕ (ligne 1) et plusieurs divisions exactes par ϕ (ligne 3).

Cependant, prendre ϕ comme une puissance de deux tout en assurant l'existence des polynômes M et M' n'est pas évident. Dans [NP08], les auteurs ne fournissent pas le processus de génération de ces deux polynômes. Dans [EG12], El Mrabet et Gama fournissent le processus de génération pour le cas particulier $E(X) = X^n + 1$.

Dans [DDV19], Didier et al. ont proposé un processus de génération pour $E(X) = X^n - \lambda$, et $\lambda \in \mathbb{Z} \setminus \{0\}$.

Dans le même article, ils proposent également un processus de génération pour l'ensemble des paramètres de l'AMNS ainsi qu'un ensemble complet d'algorithmes pour effectuer des opérations arithmétiques et de conversion dans ce système.

1.2 Crypto-systèmes embarqués sur FPGA

La croissance rapide des applications embarquées a fait des systèmes embarqués une partie importante des produits qui apparaissent dans presque tous les domaines de nos vies : téléviseurs numériques, consoles de jeux, routeurs réseau, équipements de communication numérique, imprimantes, scanners, imprimantes multifonctions, appareils électroménagers, etc.

L'objectif des systèmes embarqués est d'effectuer un ensemble de tâches spécifiques pour améliorer la fonctionnalité des grands systèmes. En conséquence, comme ils sont intégrés dans des systèmes sur puce, ils sont généralement invisibles pour les utilisateurs. Les systèmes embarqués sont généralement constitués d'unités de traitement, d'interfaces E/S pour établir la communication avec d'autres périphériques du système extérieure et de mémoire pour sauvegarder les programmes et les données. Les caractéristiques principales des systèmes embarqués sont leur puissance de calcul et leur efficacité. L'efficacité des systèmes embarqués inclut des caractéristiques telles que le prix global, la durabilité du produit du produit et la consommation d'énergie. La combinaison unique de performances de type matériel et de flexibilité de type logiciel fait du FPGA (Field

Programmable Gate Array) une solution très prometteuse pour les systèmes embarqués. Un tel système réduira considérablement les risques de conception et augmentera la production.

1.2.1 Réseaux logiques programmables

Les réseaux de portes programmables sur site (FPGA) sont utilisés depuis longtemps car ils peuvent créer de manière flexible des conceptions reprogrammables pour le matériel physique, améliorant ainsi les performances des applications. Pour les applications de haut niveau dont les paramètres changent fréquemment, l'utilisation de circuits intégrés spécifiques à l'application (ASIC) devient un fardeau en raison de l'incapacité de mettre en œuvre des changements fonctionnels dans la conception de son matériel. Les FPGA offrent aux développeurs la possibilité de créer des applications qui peuvent être modifiées et reprogrammées sur la carte à base de Bloc logique configurable (CLB), pour obtenir une accélération matérielle quasi réelle sans avoir à fabriquer et redéployer des puces ASIC physiques.

L'architecture de base du Xilinx FPGA est un tableau bidimensionnel d'éléments logiques numériques, ces éléments logiques sont regroupés en blocs logiques configurables (CLB). Chaque CLB se compose de déclencheurs et de tables de consultation (LUT). Ces CLB sont reliés entre eux par des interconnexions programmables et des matrices de commutation. Les FPGA modernes ont de nombreuses améliorations, parmi lesquelles des blocs de mémoire plus grands, Block RAM (BRAM) et Ultra RAM (URAM). Ces éléments de stockage plus volumineux sont denses et rapides et ont de nombreuses applications. Les FPGA modernes ont également un module arithmétique intégré (DSP), capable de faire les opérations de multiplication, d'addition / soustraction de 17×17 et d'autres fonctions logiques. Nous utilisons des multiplieurs de taille 17×17 parce que les opérateurs sont plus faciles à manipuler, d'une part. D'autre part par ce qu'à la base les DSP sont des multiplieurs asymétriques de taille (27×18) bits. Une grande partie du circuit FPGA est dédiée aux CLBs. Un CLB (Configurable Logic Module) est composé de LUT et de bascules. Les blocs d'E / S sont organisés par zone et prennent en charge diverses normes d'interface. Le DSP48E2 est un circuit logique de traitement de signal numérique qui prend en charge l'arithmétique et peut également être utilisée pour la détection de forme, etc.

Il est évident d'après le tableau 1.1 que nous avons beaucoup plus de LUT que de circuits DSP. Bien que la fonction DSP Slice (DSP48E2) ait un accumu-

FPGA	DSP	LUTs (slices)	BRAM (Kb)
Virtex® 7 (XC7V2000T)	2,160	2,443,200	1,292
Kintex® 7 (XC7K480T)	1,920	2597,200	955
Artix® (XC7A200T)	740	269,200	365

TABLE 1.1 – Ressources disponible dans les FPGA Virtex 7, Kinex7 et Artex 7

lateur de 48 bits et un multiplicateur (27×18) bits, nous essayerons de configurer ces LUT pour avoir une implantation cryptographique plus performante et plus robuste. Cela peut être fait par la réutilisation des ressources pour de multiples opérations de calculs similaires dans l'implantation de l'algorithme. Toutefois, cela peut entraîner des implantation plus lentes, car cela réduit le parallélisme. En pratique, il y a un compromis entre l'utilisation de la surface et le temps de calcul de l'implantation.

1.2.2 État de l'art sur les architectures d'implantation d'un crypto-système

Nous trouvons dans la littérature plusieurs architectures d'implantation d'un crypto-système. En effet peu après l'apparition du chiffrement à clé publique, plusieurs travaux qui traitent des architectures pour la multiplication modulaire des grandes entiers sont apparus. Ces architectures comprennent à la fois des implémentations matérielles et logicielles pour le système sur puce, comme pour les gros ordinateurs. La plupart de ces travaux sont basées sur la multiplication modulaire de Montgomery (noté MMM) [Mon85] combiné avec l'un ou plusieurs techniques d'optimisation. Il s'agit, par exemple des architectures systolique ([Atr65][Eve90][DL92][BP01][BÖPV03][Kor94][TK99][WCSG03]), l'architecture Cox-Rower ([BM14], [KYKS16]), l'architecture en Co-design ([Šim03], [OD04] [ŠFD03]), l'architecture Logicielle ([LHL03] [KAK96]) et l'architecture CASM (Channel-based Algorithmic State Machine, en anglais) [OD04].

1.2.2.1 Architectures Systoliques

Ce type d'architecture est basée sur un réseau systolique, qui est généralement défini comme un ensemble de processeurs élémentaires (PE) identiques qui échangent et calculent des données régulièrement. Chaque processeur reçoit les données en provenance des processeurs voisins, ensuite il effectue le calcul

arithmétique, puis il envoie le résultat au processeur suivant. Ces processeurs évoluent en parallèle, sous le contrôle de la même horloge globale comme le montre la figure 1.1, [QR89].

Cela fait de l'architecture systolique une solution adéquate pour les problèmes du grand nombre de calculs dans l'arithmétique des grands entiers.

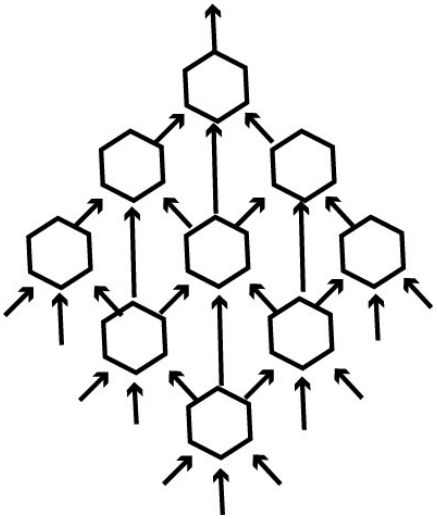

FIGURE 1.1 – Architecture Systolique

1.2.2.2 Architectures Cox-Rower

L'architecture Cox-rower a été présentée par Kawamura et al. [KKSS00]. La figure 1.2 représente une version de l'architecture cox-rower.
Ce type d'architecture est lié à la nature des représentation utilisées den RNS. Il existe une unité Cox et n ensembles d'unités Rower. Chaque unité Rower comprend un accumulateur et un multiplicateur dont la réduction modulaire par a_i ou bi. L'unité Cox est constituée d'un accumulateur, d'un additionneur calculant q. Il calcule bk bit par bit. L'unité Cox fonctionne de manière à diriger les unités Rower qui effectuent la plus grande partie de la multiplication de Montgomery.

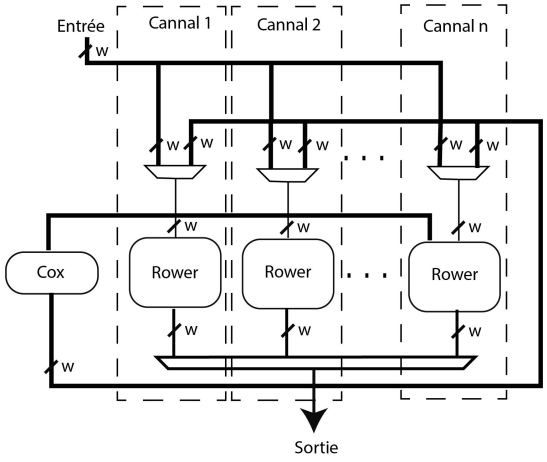

FIGURE 1.2 – Architecture Cox-Rower de [Gui10]

1.2.2.3 Architectures Co-design

La conception matérielle/logicielle offre de multiples possibilités pour accélérer l'ECC à un prix modéré, en diminuant la surface de silicium. Une étude de la littérature récente permet d'identifier trois méthodes de co-design pour implanter l'ECC.

La première méthode vise à implanter l'opération de doublement et d'addition en matérielle et les opérations qui restent en logiciel. Dans [JTB+01] Janssens et al. ont implanté l'ECC en co-design. Pour cela, les opérations arithmétiques ont été implantées matériellement. Les résultats intermédiaires ont été stockés dans la mémoire RAM. Afin d'être utilisé par la machine d'état pour contrôler les opérations d'addition/doublement de point. Bien que cette méthode est très rapide, le coût matériel est élevé.

La deuxième méthode consiste à implanter les opérations arithmétiques en matérielle. Tant dit que toutes les autres opérations, c'est-à-dire l'addition/ doublement et multiplication scalaire, ont été implanté en logiciel.
D'une manière générale, cette méthode offre une grande flexibilité, y compris la possibilité d'intégrer les dernières contre-mesures contre les attaques par canaux cachés. D'autre part, cette méthode nécessite un espace de stockage important pour les variables intermédiaire utilisés par le matériel [MSFUCAB11].

1.3 Conclusion

Dans ce chapitre, nous avons introduit en premier lieu la multiplication modulaire dans le système de représentation AMNS, ainsi que la multiplica-

tion modulaire dans RNS et la multiplication modulaire de Montgomery CIOS. Ensuite, nous avons répertorié les différentes architectures utilisées dans la littérature afin d'implémenter des systèmes cryptographiques.

2 Conceptions et implantation haut niveau de la multiplication modulaire en AMNS

Sommaire

Ce chapitre présente la méthodologie et les outils utilisés pour l'implantation de nos architectures sur un SOC hybride. Le but est de concevoir des architectures performantes dédiées aux arithmétiques modulaires dans le système AMNS, ensuite d'évaluer les performances, ressources selon les résultats présentés dans la littérature. Notre plateforme cible est la carte Zedboard, basée sur le SOC $XC7Z020-1CLG484C,\ Zynq-7000$ [Xil14b]. Tout au long de ce chapitre, nous utiliserons les termes "ZedBoard". Ainsi que la "plate-forme Zynq 7000" de manière interchangeable. La plate-forme de développement et d'évaluation Zynq 7000 (EPP) est un nouveau système sur puce (SoC System On Chip) introduit par Xilinx en 2012. Il intègre un processeur Cortex-A9 double cœur, un FPGA de la série Xilinx 7000 [PL12] et quelques périphériques courants sur une seule puce comme le montre la figure 2.1. Le processeur Cortex-A9 à double cœur convient aux applications générales et la logique programmable permet aux utilisateurs de développer de nouveaux périphériques, des accélérateurs matériels ou des unités de traitement spécifiques à une application. Le processeur utilise une logique programmable pour échanger avec un accélérateur matériel personnalisé sur le FPGA via le bus AXI (Advanced Extensible Interface).

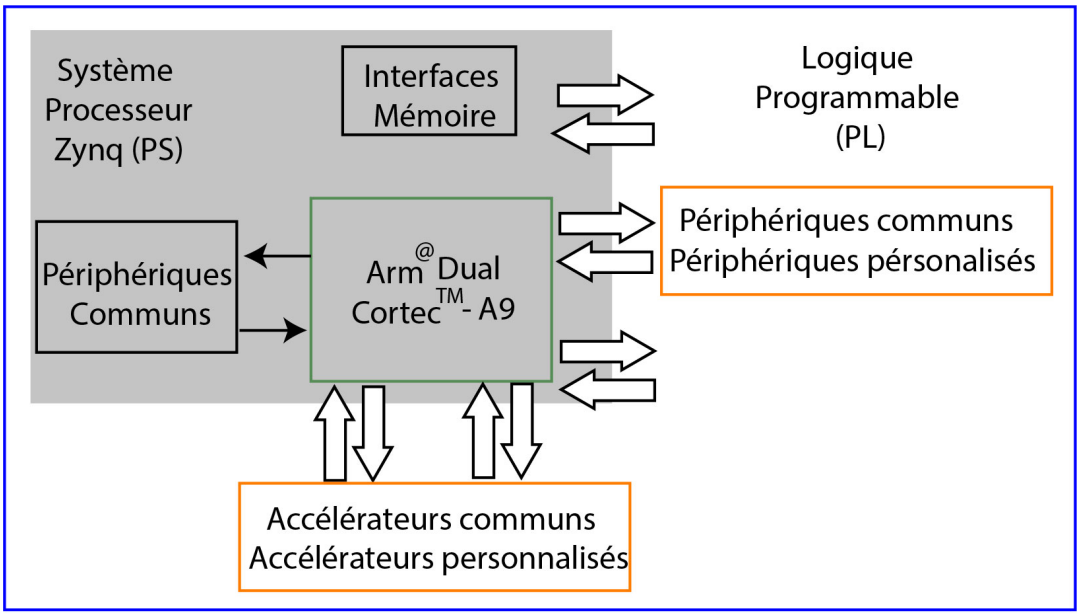

FIGURE 2.1 – Architecture Hybride de la Zedboard employée pour les SOC [CEES14]

Nous présentons l'environnement de nos implantations dans la partie 2.1. Ensuite, nous présentons d'abord notre architecture et notre implantation via

une synthèse de haut niveau et une description de RTL. Dans la section 2.2, nous avons présenté en détail les différentes techniques de pipeline de l'outil HLS que nous utilisons pour la multiplication modulaire AMNS et CIOS. Ensuite, dans la partie 2.2.3.3 nous intégrons ces IP (propriété intellectuelle) dans un environnement SW/HW. Enfin, dans la partie 2.3 nous comparons les résultats d'implantation de la multiplication modulaire d'AMNS et de CIOS. Le but est d'utiliser ces accélérateurs dans des primitives cryptographiques asymétriques.

2.1 Environnement d'implantation

Nous utilisons le logiciel Vivado HLS pour explorer de nombreuses améliorations dans l'algorithme de réduction modulaire. L'implémentation de l'algorithme de multiplication modulaire sur la carte zedboard nécessite trois étapes (voir figure 2.2) :

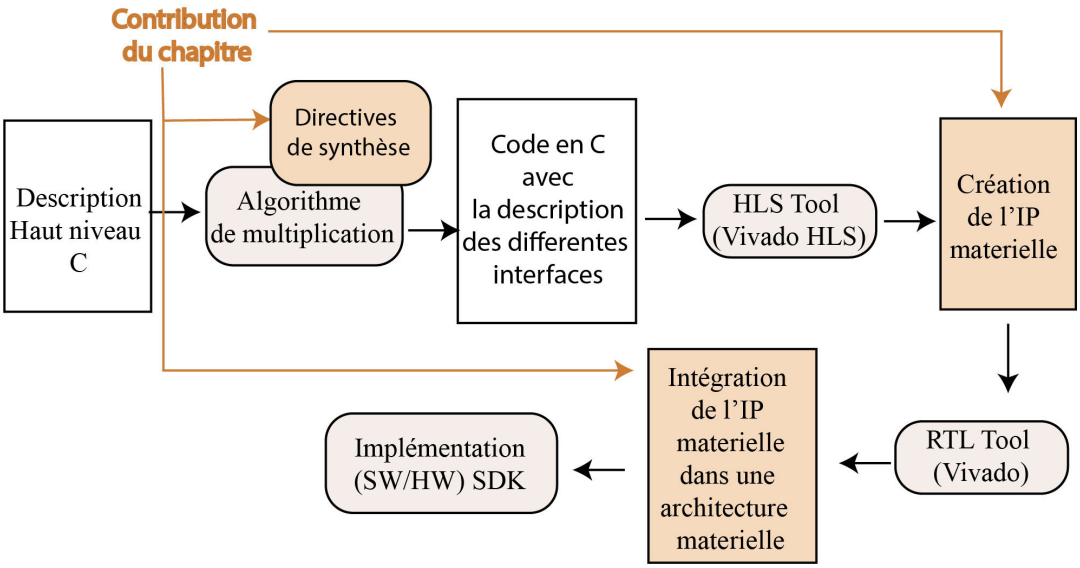

FIGURE 2.2 – Conception des étapes de compilation

— **Création de l'IP et la génération du RTL :** La première étape consiste à créer l'IP (Intellectual Property) spécifique à notre multiplication modulaire. Dans le but que les concepteurs d'implantation matérielle peuvent l'explorer dans la cryptographie asymétrique. Ensuite, de générer le RTL de L'IP et estimer les ressources et les délais utilisés.
Le logiciel utilisé dans cette étape est l'outil Vivado HLS.

— **Intégration de l'IP dans l'architecture :** La deuxième étape nécessite l'utilisation du logiciel Vivado Design Suite au lieu du logiciel Vivado HLS.

Cette étape consiste à connecter l'IP avec le bloc mémoire BRAM, DMA, les contrôleurs BRAM. Finalement nous générons le bitstream.

— **Implémentation sur la Zedboard :** Dans la dernière étape, nous utilisons le logiciel Xilinx SDK, afin de tester et exécuter l'architecture matérielle de l'algorithme de la multiplication modulaire sur la carte Zedboard.

2.2 Créations des IPs et directives de synthèse

L'outil Vivado HLS synthétise les fonctions C, C++ et System-C en blocs au sein de la hiérarchie RTL. Les fonctions de haut niveau sont synthétisées, selon le cas, en des unités avec de signaux de commande adéquats. L'outil HLS permet aux concepteurs matériels de faire des analyses et des améliorations de la conception. Il crée une implantation RTL de la conception haut niveau, en se basant sur la configuration par défaut des caractéristiques comportementales, des contraintes et de toute directive d'optimisation. Ensuite, l'outil produit un ensemble de comptes rendus récapitulatifs qui sont exploités pour évaluer l'implémentation et appliquer par la suite plusieurs optimisations pour donner aux contraintes de conception.

La conception des systèmes et des architectures matérielles se fait en général en langage HDL (Hardware Description Language) qui modélise le processus en primitives RTL (Register Transfer Level). D'autre part, les langages de haut niveau permettent de modéliser efficacement les systèmes et les applications avec moins de ressource et une meilleure configurabilité. Afin de réduire le temps de conception nécessaire, des outils de synthèse de haut niveau sont utilisés pour transformer les systèmes modélisés de langage C dans un langage de haut niveau en une implémentions RTL qui peut être synthétisée en FPGA [DZS14]. Le HLS permet non seulement de réduire le temps de conception et de débogage du langage HDL, mais aussi d'offrir une certaine souplesse dans l'implémentation matérielle finale afin de répondre aux exigences de la conception.

L'outil HLS Xilinx Vivado [Fei12] transforme une conception de spécification C, C++ et System-C, dans une conception RTL. L'outil passe par plusieurs phases pour obtenir le design. L'outil HLS planifie les opérations logiques pour chaque cycle d'horloge et affecte des ressources au matériel pour chaque opération planifiée. Pendant la conception, la logique de commande est extraite pour réaliser une machine à états finis qui enchaîne les opérations dans la description RTL.

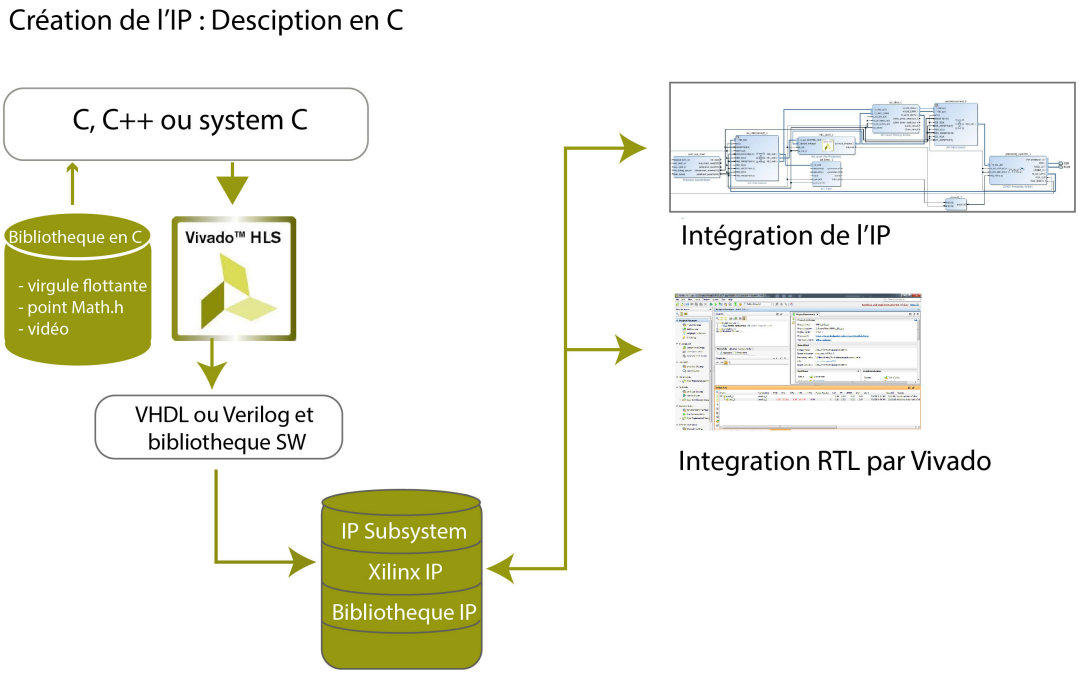

FIGURE 2.3 – Flot de conception en HLS

La synthèse de haut niveau (HLS) donne la possibilité au concepteur de structurer son programme ou de faire appel à l'outil pour réaliser des optimisations architecturales, comme le partitionnement des données, la mise en parallèle et le flux [MS09]. Ceci lui donne la possibilité de générer rapidement des conceptions différentes [MAGK16]. Les algorithmes de cryptographie sont couramment impliqués dans le but d'améliorer le débit et les performances [HAHH06]. Cela a tout naturellement conduit à des études sur la manière dont les différents systèmes cryptographiques agissent sur les performances, la consommation d'énergie et à l'utilisation des ressources [HG14].

2.2.1 Méthodologie de synthèse de l'algorithme en C

Nous décrivons dans cette partie la stratégie que nous avons appliqué sur l'algorithme de multiplication modulaire par CIOS et l'algorithme de multiplication modulaire dans le système de représentation AMNS. En effet, nous travaillons d'abord sur l'algorithme initial décrit en C, ensuite nous appliquons une ou plusieurs optimisations. Puis nous faisons la synthèse avec l'outil Vivado HLS, afin de trouver un meilleurs ADP (Area Delay Product) et throughput (débit) de l'implantation.

De ce fait, le choix de la méthode d'optimisation dépend des ressources disponibles dans le FPGA. En fait, l'intégration de pipeline peut en effet améliorer

le temps d'exécution, mais elle nécessite également l'utilisation de plus de mémoire, de DSP, de LUT et de ressources de registre. Par la suite, nous essayons de choisir la meilleure combinaison d'optimisation et d'implémentation. Nous avons introduit la méthode que nous avons utilisée pour trouver l'algorithme d'optimisation dans la figure 2.4.

Dans cette partie, nous avons introduit la méthodologie que nous avons suivie dans la partie 2.2.1, afin d'implémenter efficacement la multiplication modulaire AMNS sur FPGA. Ensuite, nous introduisons les **directives** de synthèse utilisées dans l'implantation de haut niveau dans la section 2.2.2. Dans les parties 2.2.4 et 2.2.3, nous proposons deux architectures matérielles, pour différents tailles du module $p = 128, 256, 512, 1024$ et des tailles de mots $w = 18, 36, 64$, afin de créer des accélérateurs matériels respectivement pour la multiplication modulaire CIOS et dans AMNS. Nous décrivons le choix des directives de synthèse utilisées dans les deux algorithmes de multiplication modulaire CIOS et la multiplication modulaire dans le système de représentation AMNS.

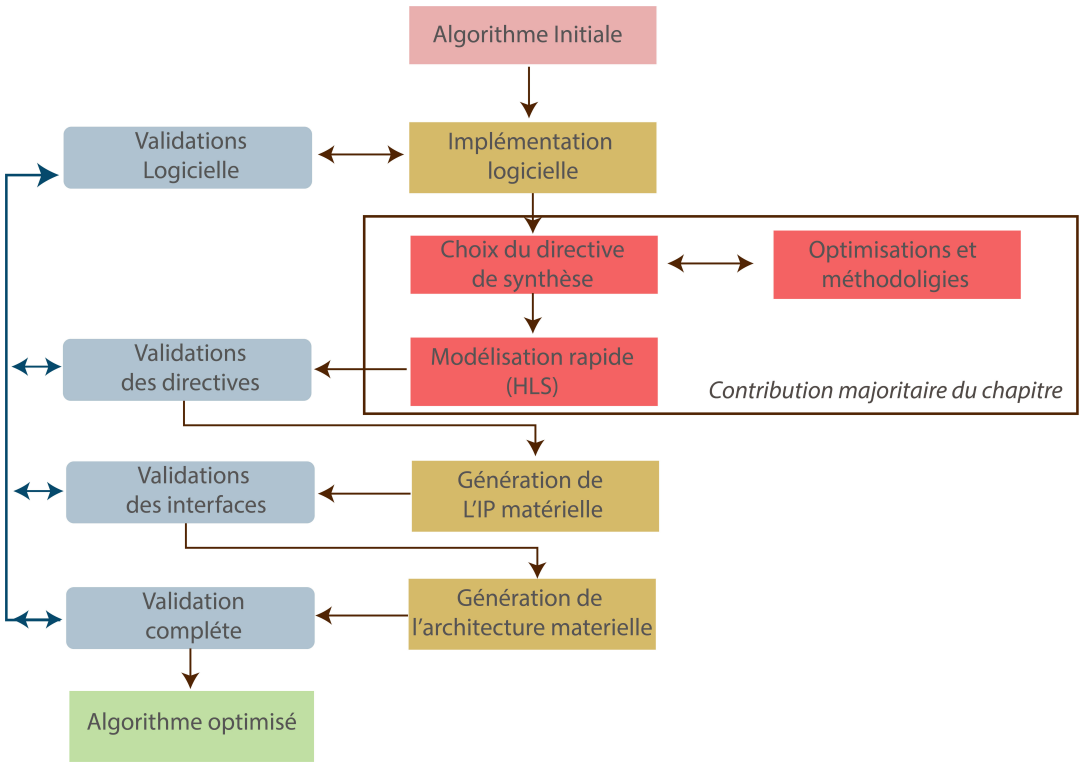

FIGURE 2.4 – Méthodologie de synthèse

2.2.2 Directives de synthèse Vivado HLS

Vivado HLS dispose d'un catalogue de directives de synthèse qui peuvent être associées à des parties du code source du logiciel. Ces directives spécifient les méthodes de synthèse que l'outil doit utiliser sur une partie de code donnée. Dans cette partie, nous allons introduire les directives les plus importantes que nous avons utilisés. La liste complète des directives est disponibles dans [Xil14a].

On peut placer ces directives soit :

— Sous la forme d'une fonction par *#pragma*, on le met directement dans le code source.

— soit dans un fichier spécifique pour les directives, qui sera associé dans le projet de chaque solution.

Les directives sont utilisées selon les fonctions qu'on introduit dans le programme. Afin d'optimiser au mieux les ressources et le temps d'exécution utilisé.

2.2.2.1 Directive UNROLL

La directive UNROLL s'explique d'elle-même. Elle s'applique aux boucles et elle comporte un paramètre spécifique qui spécifie le degré du parallélisme d'une boucle déroulée. Un facteur de 0 ou 1 laisse la boucle inchangée. Si aucun facteur n'est spécifié, la boucle sera entièrement déroulée. Cette directive est utilisée afin de réduire le nombre de cycles en déroulement de boucle. En contrepartie le nombre des ressources sera répliqué, y compris les registres de mémoire. Prenons l'exemple suivant :

```
for(i=0; i<4; i++)
c[i] = a[i] * b[i];
```

Lorsque la boucle est répété sans directive, l'architecture matérielle utilisée sera composée d'un multiplicateur et un bloc de mémoire pour chaque itération, et chaque itération exige un cycle d'horloge, de sorte qu'au moins quatre cycles d'horloge sont nécessaires pour l'implémentation.

Mais si nous utilisons la directive Unroll et si nous prenons le facteur de directive égale à 4, alors pour chaque cycle d'horloge, quatre opérations de lecture et quatre opération d'écriture et quatre multiplicateur peuvent s'exécuter en un seul cycle d'horloge.

La directive UNROLL est utilisée pour spécifier combien de cycles doivent être exécutés à chaque itération. Toutefois, cette directive doit être utilisée avec précaution. Si le facteur de déroulement est fixé à un niveau supérieur à celui requis, une quantité importante des ressources matérielles inutiles pourrait être consommée.

2.2.2.2 Directive PIPELINE

C'est les dépendances à l'intérieur d'une boucle, lorsque cette directive s'applique aux boucles, elle permet d'exécuter en parallèle des itérations de boucles consécutives. En fait, le principe est de diviser en une succession d'opérations les instructions de la boucle, et d'exécuter en décalé chaque étage par de nouvelles itérations. Ceci est illustré dans la figure 2.5 . Elle montre deux exemples de la manière dont une boucle contenant quatre opérations consécutives (OP1, OP2, OP3 et OP4) peut être mise en pipeline. Dans le cas idéal, la prochaine itération en boucle peut commencer un cycle après le début de l'itération précédente. Cela signifie que le matériel utilisé pour réaliser l'OP1 ne sera jamais inactif. Cependant, si l'OP1 dépend de la sortie de l'OP2 de l'itération précé-

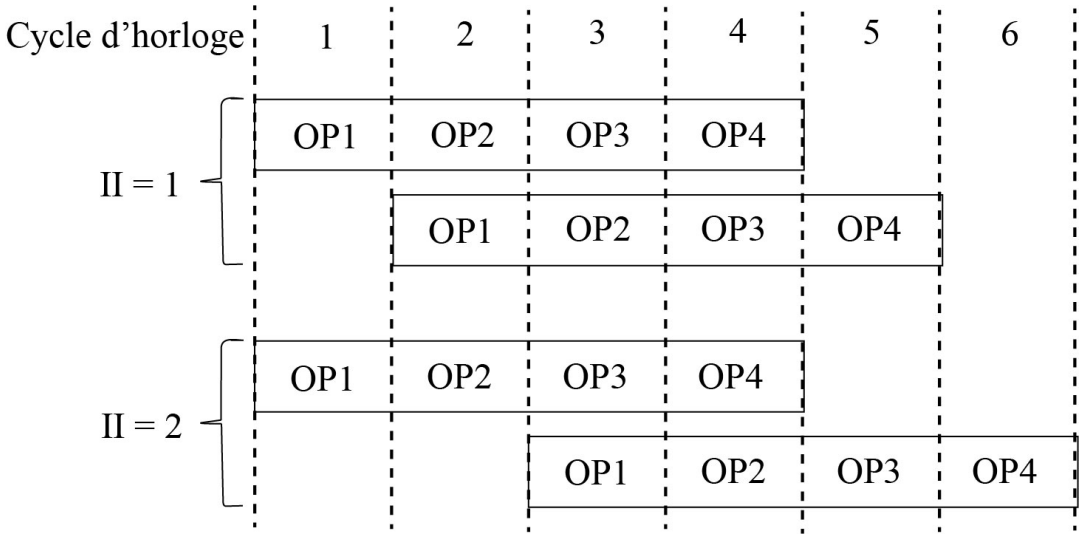

FIGURE 2.5 – Exemple de directive PIPELINE

dente, il faudra peut-être attendre deux cycles avant qu'il puisse commencer. Le cas idéal est celui d'un intervalle d'initiation (II) de 1, tandis que l'autre cas est celui d'un II de 2.

2.2.2.3 Intervalle d'initiation (II)

C'est le nombre de cycles d'horloge nécessaire à l'alimentation incrémentale des données d'entrées. La directive PIPELINE permet de spécifier l'intervalle d'initiation (II) souhaité. Si l'intervalle d'initiation (II) souhaité ne peut pas être accompli, l'outil incrémentera le II jusqu'à ce qu'il soit fonctionnel. Par défaut, l'intervalle d'initiation (II) est fixé à 1. Lorsque la directive PIPELINE est appliquée à une boucle ou à une fonction, toutes les sous-boucles seront entièrement déroulées. Ce n'est pas idéal si les sous-boucles sont importantes. Dans ces situations, la directive DATAFLOW devrait être envisagée. Une description plus détaillée des options est disponible dans [Xil14a].

2.2.2.4 Directive DATAFLOW

Les directives au niveau de la boucle ou de la hiérarchie des fonctions à laquelle la directive DATAFLOW est placée sont désignés comme une "*Région de flux de données*". À l'intérieur d'une région de flux de données, les boucles et les appels de fonctions consécutives qui accèdent aux mêmes structures de mémoire sont regroupés en "*Directive de flux de données*". Ces directives de données sont reliées entre elles par des canaux FIFO (First In First Out). Toutes les directives de flux de données fonctionnent indépendamment les uns des autres. Chaque canal FIFO met en file d'attente les données de sortie d'une donnée et l'envoie au processus suivant de traitement des données lorsque ce dernier est prêt à être saisi. Cela permet à toutes les directives de flux de données d'être exécutées simultanément.

Lorsque la directive DATAFLOW n'est pas utilisée, la logique de boucle devient inactive dans le circuit matériel lorsqu'il a terminé sa tâche et qu'il attend les boucles qui suivent se terminent. Dans cette situation, le temps complet d'exécution de la fonction est égal à la somme des temps d'exécution de chaque processeur de boucle. Cependant, lorsque la directive DATAFLOW est utilisée dans une situation de flux de données, le temps d'exécution de la fonction devient égal au temps d'exécution du processeur de boucle le plus lent.

2.2.3 Génération de l'IP matérielle par HLS de l'algorithme de la multiplication modulaire en AMNS

Dans [CDD+19], nous avons proposé une architecture de la multiplication modulaire dans le système AMNS [DDV20]. À notre connaissance, cette im-

plantation n'a jamais été faite dans le système AMNS.

Comme expliqué dans la partie 1.1.3, la multiplication modulaire dans l'AMNS se fait sur trois opérations avec trois degré de dépendance séquentielle ; une multiplication polynomiale, ensuite une réduction externe suivie d'une réduction interne. À cause de ces dépendances, certains étages *slice* DSP sont passifs lors de certains cycles, ce qui diminue la latence de l'implantation. Ce pour cela, nous choisissons de les configurer dans le pipeline pour améliorer l'efficacité des implantations matérielle.

Dans cette partie, nous décrivons en premier lieu l'architecture de notre IP matérielle pour la multiplication modulaire dans le système AMNS. Ensuite, nous présentons les différentes directives d'optimisation et les modifications algorithmique que nous avons apportées sur l'algorithme initial.

2.2.3.1 Architecture matérielle de l'IP

La figure 2.6 représente l'architecture proposée. Nous avons implémenté plusieurs versions pour des différentes largeurs de mot, comme $w = 32, 36, 54$ et 74, aussi pour de différentes tailles de p $(128, 256,\ 512, 1024)$. Nous décrivons l'architecture pour un modulus $p = 128$ bits et nous notons le nombre de coefficients du polynôme $n = 4$ pour $w = 32$ bits. Ensuite, nous adaptons la même architecture pour les autres tailles de modules.

La multiplication modulaire AMNS s'applique en trois étapes :

— **Étape 1** : la multiplication polynomial de A par B pour $n = 4$, afin de déterminer le polynôme résultant $C = A \times B$ de n^2 coefficients.

— **Étape 2** : nous appliquons la réduction externe afin de maintenir le degré des éléments de l'AMNS strictement inférieur à n. En utilisant le polynôme de réduction externe E tels que $C' = C \mod E$, et le polynôme $E = X^n - \lambda$ et $\lambda = \pm 2^h$ avec $h \in \mathbb{N}$. Le résultat sera stocké dans le registre C' avec $n = 4$.

— **Étape 3** : Nous utilisons un algorithme de réduction interne pour nous assurer que la norme infinie des éléments AMNS est strictement inférieure à ρ. Ici, nous avons appliqué l'algorithme 9.

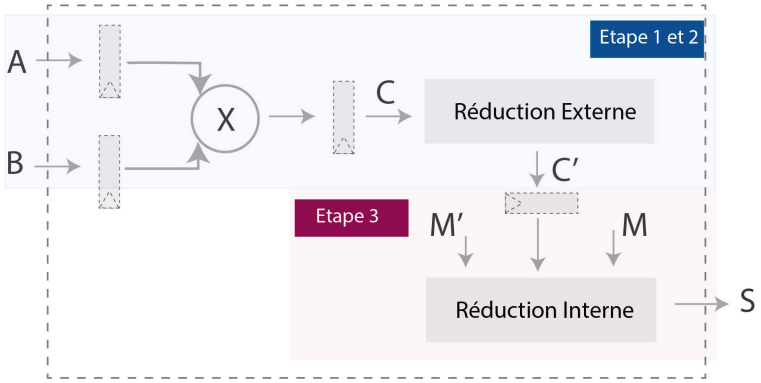

FIGURE 2.6 – Architecture de la multiplication modulaire dans le système de représentation AMNS

A) **Architecture des étapes 1 et 2**

Nous avons remarqué que les étapes 1 e t 2 de la multiplication modulaire dans AMNS peuvent être combinées. En effet, le produit $C = A \times B$ et $C' = C \mod E$ peuvent être combinés comme suit :

- $c'_0 = a_0b_0 + \lambda(a_1b_3 + a_2b_2 + a_3b_1)$,
- $c'_1 = a_0b_1 + a_1b_0 + \lambda(a_2b_3 + a_3b_2)$,
- $c'_2 = a_0b_2 + a_1b_1 + a_2b_0 + \lambda(a_3b_3)$,
- $c'_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$.

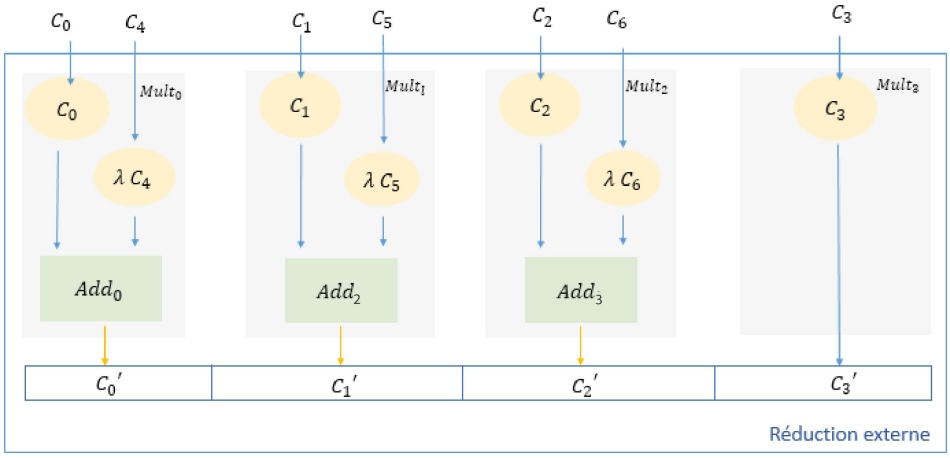

FIGURE 2.7 – Réduction externe

Comme le montre la figure 2.7, les coefficients du polynôme C' peuvent être calculés en parallèle. Tels que :

· $c_0 = a_0b_0$

· $c_4 = a_1b_3 + a_2b_2 + a_3b_1$,

· $c_1 = a_0b_1 + a_1b_0$

· $c_5 = a_2b_3 + a_3b_2$,

· $c_2 = a_0b_2 + a_1b_1 + a_2b_0$

· $c_6 = a_3b_3$,

· $c_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$.

L'architecture que nous proposons dans cette étape est composée de quatre cellules $Mult\_i$. Chaque cellule $Mult\_i$ est composée d'un $sliceDSP$ et d'un additionneur Add_i. L'idée ici est de calculer tous les coefficients du polynôme C simultanément dans le même cycle d'horloge, ensuite d'exécuter les cellules Add_i en parallèle dans un deuxième cycle d'horloge. La cellule $Mult_i$ est composée d'une multiplication est d'une addition.

B) **Architecture de l'étape 3**

L'étape 3 est l'opération la plus complexe de l'algorithme de multiplication modulaire. En fait, elle est composée de trois opérations qui sont détaillées dans l'algorithme 9.

Dans cette étape, nous présentons l'architecture matérielle de la réduction des coefficients, nous utilisons la méthode $Montgomery - like$ qui nécessite ces constantes suivantes en entrée :

— Un entier $\phi \in \mathbb{N}\backslash\{0\}$

— Un polynôme $M \in \mathbb{Z}_{n-1}[X]$, tel que : pour un système AMNS $\mathcal{B} = (p, n, \gamma, \rho, E)$ on a $M \in \mathcal{B}$, $M(\gamma) \equiv 0 \pmod{p}$

— Un polynôme $M' \in \mathbb{Z}_{n-1}[X]$ et $M' = -M^{-1} \mod (E, \phi)$

— La valeur de ϕ doit être choisi comme une puissance de deux, pour que l'étape précédente soit efficace.

L'architecture matérielle de cette étape est représentée dans la figure 2.8, pour un $p = 128$ et $n = 4$. Elle est composée de trois opérations suivantes :

— La première opération, est la détermination du polynôme $Q = (C' \times M') \mod (E, \phi)$. Nous avons comme entrées les coefficients des polynômes C', $M' \in \mathcal{B}$. Nous utilisons quatre cellules $Mult\_c$ et quatre cellules Add_c. Nous appliquons la même méthode utilisée dans l'étape 1 et 2. En effet, nous avons combiné l'étape de la multiplication avec l'étape de réduction externe. Par la suite, nous calculons les produits intermédiaires de tous les coefficients dans le même cycle d'horloge, en suite nous exécutons les cellules Add_c dans un deuxième cycle d'horloge afin d'avoir en sortie les coefficients du polynôme Q, qui seront utilisés dans l'étape suivante en tant que entrée.

— La deuxième étape, consiste à calculer le polynôme $R = Q \times M \ mod \ E$. Les entrées sont les coefficients des polynômes Q et M. Nous utilisons quatre cellule $Multq\_i$, quatre cellule de $Addq\_i$ et quatre cellule de $Addr\_i$.

• Les cellules de $Multq_i$ et $Addq_i$ sont utilisées pour calculer la multiplication $(Q \times R) \ mod \ E$, qui sera fourni à l'étape suivante.

• La cellule $Addr_i$ est utilisée pour calculer la somme polynomiale $C' + (Q \times M)$ le résultat est enregistré dans le polynôme R qui sera fourni comme entrée dans l'étape suivante.

— L'étape finale est une division $S = R \backslash \phi$. Nous utilisons quatre cellules de $shift\_right$ puisque ϕ est un entier en puissance de 2, et le résultat est enregistrer dans le polynôme S.

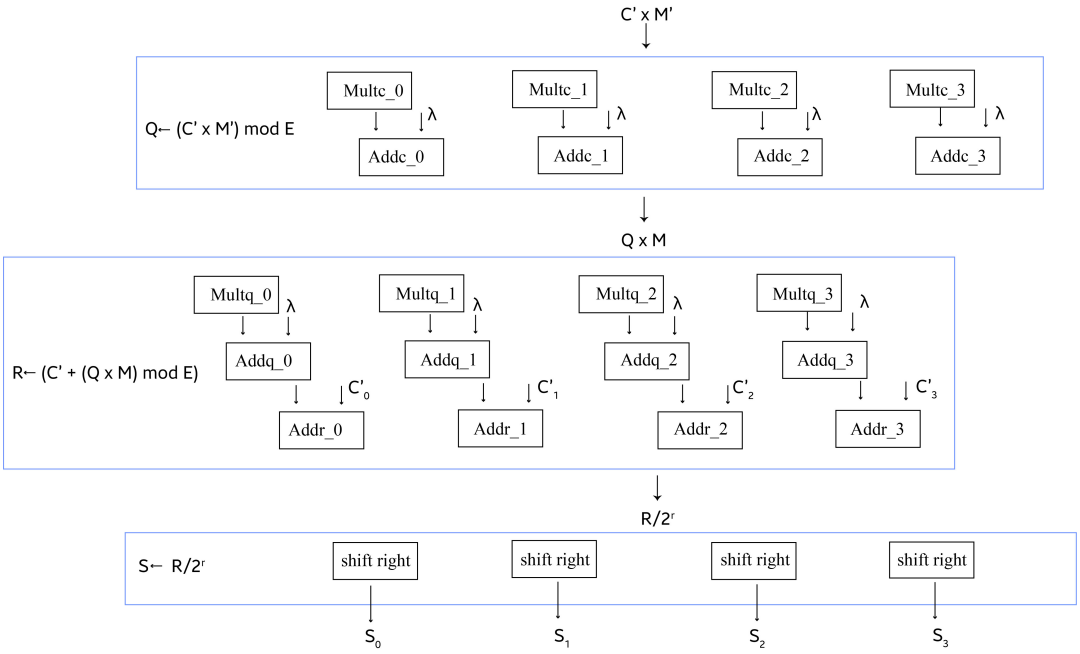

FIGURE 2.8 – Réduction interne

2.2.3.2 Synthèse de l'algorithmique et directive de synthèse

Dans cette partie nous présentons en premier lieu les modifications que nous avons apportées sur l'algorithme initial. Ensuite, nous présentons les directives de synthèse que nous avons choisies et nous intégrées dans l'algorithme initiale. L'algorithme de référence (Algo 10), qui correspond à la multiplication modulaire dans le système de représentation AMNS.

A) **Explication de l'algorithme**

La multiplication des polynômes de degré $n-1$ dans AMNS est assuré

Algorithm 10 Algorithme de référence

Entrées: $A, B \in \mathcal{B}$ et $M, M' \in \mathbb{Z}_n[X]$

Sorties: $S \in \mathcal{B}$

1: **pour** $i = 0 \dots n-1$ **faire**
2: $\quad c'_i = mult\_mod\_poly\_c(a_i, b_i)$
3: **fin pour**
4: **pour** $i = 0 \dots n-1$ **faire**
5: $\quad q_i = mult\_mod\_poly\_q(c'_i, m'_i)$
6: **fin pour**
7: **pour** $i = 0 \dots n-1$ **faire**
8: $\quad q'_i = mult\_mod\_poly(q_i, m_i)$
9: **fin pour**
10: **pour** $i = 0 \dots n-1$ **faire**
11: $\quad r_i = add\_poly\_q(c'_i, q'_i)$
12: **fin pour**
13: **pour** $i = 0 \dots n-1$ **faire**
14: $\quad r_i = shift\_right(r_i)$
15: **fin pour**

par la fonction $mult\_mod\_poly_c$ qui est une multiplication polynomiale classique. Cependant, le polynôme résultant aura un degré de $2n-2$ que nous devons réduire modulo E (le polynôme de réduction externe). Par la suite, nous utilisons les fonctions de la réduction interne afin d'obtenir le résultat dans $\mathcal{B}$. Les fonctions que nous avons employé dans l'algorithme sont les suivants :

— La fonction $mult\_mod\_poly\_c$ qui permet de multiplier les deux polynômes A et B comme suit :

- $c'_0 = a_0b_0 + \lambda(a_1b_3 + a_2b_2 + a_3b_1)$,
- $c'_1 = a_0b_1 + a_1b_0 + \lambda(a_2b_3 + a_3b_2)$,
- $c'_2 = a_0b_2 + a_1b_1 + a_2b_0 + \lambda(a_3b_3)$,
- $c'_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$.

— La fonction $mult\_mod\_poly\_q$ permet de calculer $Q = C' \times M' \ mod \ E$ comme suit :

- $Q_0 = c'_0m'_0 + c'_1m'_3 + c'_2m'_2 + c'_3m'_1$,
- $Q_1 = c'_0m'_1 + c'_1m'_0 + c'_2m'_3 + c'_3m_2$,
- $Q_2 = c'_0m'_2 + c'_1m'_1 + c'_2m'_0 + c'_3m_3$,
- $Q_3 = c'_0m'_3 + c'_1m'_2 + c'_2m'_1 + c'_3m'_0$.

— La fonction *mult_mod_poly_y* permet de calculer $Q' = Q \times M \ mod \ E$ comme suit :

- $q'_0 = q_0m_0 + q_1m_3 + q_2m_2 + q_3m_1$,
- $q'_1 = q_0m_1 + q_1m_0 + q_2m_3 + q_3m_2$,
- $q'_2 = q_0m_2 + q_1m_1 + q_2m_0 + q_3m_3$,
- $q'_3 = q_0m_3 + q_1m_2 + q_2m_1 + q_3m_0$.

— La fonction *add_poly_q* permet de calculer $R = C' + Q' \ mod \ E$ comme suit :

- $R_0 = q'_0 + c'_0$,
- $R_1 = q'_1 + c'_1$,
- $R_2 = q'_2 + c'_2$,
- $R_3 = q'_3 + c'_3$.

— Finalement, la fonction $shift\_right$ permet de calculer $S = R \backslash \phi$. En effet, ϕ est une puissance de deux ($\phi = 2^h, h \in \mathbb{N}$). Cette fonction est tout simplement un décalage à droite de h bit.

B) **Problématique d'optimisation**

La première analyse de l'algorithme de référence nous permet de poser les remarques suivantes :

— Les coefficients des polynômes A, B, M, M', Q, Q', R et S varient respectivement entre w et $2w$. Afin de mieux utiliser le module DSP, nous choisissons que la taille de w soit un multiple de 18 bits (36, 54 et 72 bits).

— L'algorithme contient cinq boucles consécutives, chaque boucle a n itérations, où n est le degré des polynômes A, B et S.

— Le calcul des coefficients c'_i, q_i, q'_i nécessite d'accéder à la mémoire BRAM en mode lecture et en écriture à chaque itération.

— Le calcul des coefficients r_i du polynôme R nécessite l'utilisation d'un accumulateur à chaque itération.

C) **Listes des directives de synthèse**

Directive de synthèse 1 : **Parallélisme :** Nous avons ici trois fonctions qui emploie une multiplication dans chaque itération. Dont la taille dépasse les caractéristiques du DSP48, qui admet un multiplicateur de 18×24 et un accumulateur de 48 bits.
Amélioration envisagé : Segmentation de la fonction en deux fonction l'une pour la multiplication et l'autre pour l'addition. Ensuite nous appliquons la directive unroll.

Directive de synthèse 2 : **Stockage des données :** La taille des coefficients des polynômes intermédiaires est très importante, il est judicieux d'utiliser des mémoires rapides.
Amélioration envisagé : mémoire local (BRAM), déroulage ou pipeline de la boucle.

Directive de synthèse 3 : **Optimisations fines :** Les fonctions de multiplication utilisée sont impossible à segmenter.
Amélioration envisagé : L'empois de directive Dataflow.

Directive de synthèse 4 : **Lecture des donnée :** Les coefficients des entrées A et B, et les polynomes constante M, M' sont stocké dans la mémoire externe. Qui prend plus de temps dans la lecture.
Amélioration envisager : L'empois de directive AXI Stream.

D) **Exploration des directives**

1. **Parallélisme** : La première implémentation consiste à modifier le code initial en l'adaptant à la fonctionnalité des DSP. En effet, la cellule $Mult\_i$ que nous avons présenté dans la figure 2.7 contient deux fonctions l'une pour la multiplication des coefficients et l'autre pour l'addition.

   La cellule $Mult\_i$ calcule le coefficient C'_i en deux étapes ; la première est une multiplication ($C = A \times B$) et la deuxième est une addition $C'_i \leftarrow C'_i + (\lambda \times C_i)$ qui représente la réduction externe du polynôme C' ($C' = C \mod E$).

2. **Stockage des données** : Nous utilisons le contrôleur DMA (Direct Memory Access) afin de gérer la communication entre l'IP et

Algorithm 11 *Cellule Add_i*

Entrées: C,n
Sorties: $C' = (A \times B) \mod E$

```
1: #pragma PIPLINE II=1
2: #pragma UNROLL 2n
3: si i < n alors
4:     C'[i] = C'[i] + C[i]
5: sinon
6:     si i > n alors
7:         C'[i] = C'[i] + λ × C[i]
8:     fin si
9: fin si
10: return C
```

la mémoire BRAM. Ce qui implique l'utilisation du protocole AXI-Stream.

3. **Optimisations fines** : L'application des directives datafollow et unroll.

```
#pragma HLS UNROLL
#pragma HLS DATAFOLOW
```

4. **Lecture des donnée** : L'application des directives Rssource Stream.

```
#pragma HLS INTERFACE ap_fifo port=in_stream8#
   pragma HLS INTERFACE ap_fifo port=out_stream
```

2.2.3.3 Intégration de l'IP dans un environnement SW/HW

La deuxième étape, et après la génération de notre IP et identification du temps nécessaire pour notre fonction, nous avons construit un système embarqué de base et nous avons intégré notre IP matérielle avec le Vivado 2016.4. Pour rendre notre conception fonctionnelle, nous avons inclus deux interconnexions AXI, le contrôleur DMA : l'une pour les registres de contrôle, l'autre pour l'ACP (Accelerator Coherency Port), le sous-système de temporisation, qui sera utilisé pour donner des mesures très précises de la synchronisation du matériel. Ensuite, nous avons généré un script Tcl pour compléter le processus nécessaire à la connexion.

2.2.4 Génération d'une IP matérielle par HLS de l'algorithme de Montgomery CIOS

Dans cette partie, nous décrivons les étapes d'implantation de l'algorithme de Montgomery CIOS. L'implantation de l'algorithme CIOS est faite de la même façon que la multiplication modulaire dans l'AMNS. Afin de mener une comparaison équitable entre les deux implémentation dans la section 2.3.1.

Nous souhaitons créer une IP matérielle, afin qu'elle soit exploitée par les concepteurs de l'implantation matérielle de cryptographie asymétrique. Nous cherchons à avoir un compromis entre les ressources utilisées, le temps d'exécution et une architecture matérielle moins complexe. Dans cette section, nous décrivons la méthode utilisée lors de la conception du matériel. La figure 2.9 décrit le flot de conception de notre architecture.

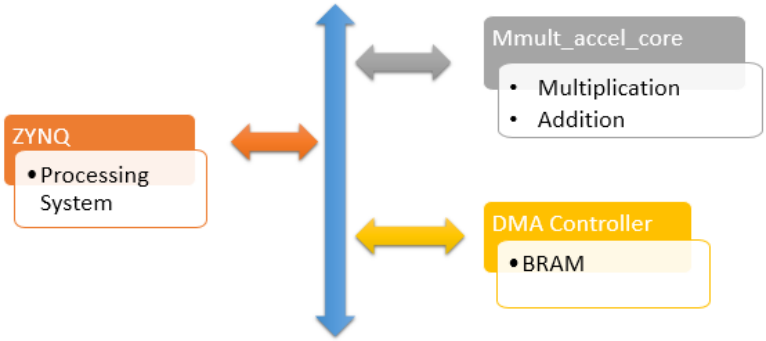

FIGURE 2.9 – Flot de conception

L'opération principale de notre IP est la multiplication modulaire de Montgomery CIOS. Il existe des dépendances hiérarchiques entre les opérations nécessaires :

— Nous avons besoin d'une fonction pour ramener les opérandes dans le domaine de Montgomery.

— Opérations de base des éléments : décalage à droite, décalage à gauche.

— Les opérations sur $\mathbb{Z}_p$: multiplication, soustraction, addition de grands entiers.

2.2.4.1 Développement de l'IP matérielle avec Vivadeo HLS

La première étape consiste à utiliser le logiciel Vivado HLS 2016.4 pour créer une architecture RTL à partir de notre code C qui est détaillé dans l'algorithme

4. Le synthétiseur transforme chaque ligne de l'algorithme en une opération matérielle et insère ces opérations dans des cycles d'horloge. En utilisant les informations de la période d'horloge, il place dans un seul cycle d'horloge, autant d'opérations que possibles. Nous avons considéré le mode Standalone (sans système d'exploitation) pour développer un IP matériel de l'algorithme qui permet d'améliorer les performances. Afin d'optimiser au mieux le développement de l'IP, nous avons utilisé les directives d'optimisation suivantes dans nos algorithmes :

A) **Directive de gestion de la mémoire : Axi-stream**
Nous utilisons deux types de données en entrée :

— La variable $carry\_data$: qui est une variable intermédiaire mise à jours après chaque opération arithmétique.

— Les entrées $a, b, carry\_data$: qui ont une taille importante, pour plus de performances et pour un stockage optimal, nous choisirons la taille de ces coefficients un multiple de 18 ; soit 36, 54,72.

Afin de gérer le stockage de donnée dans la mémoire BRAM et dans la mémoire externe nous utilisons la directive $AXI - Stream$ et la directive BRAM. En effet, la taille des opérandes est importante et augmente dans les variables intermédiaires avec la multiplication des opérandes. Nous utilisons les deux méthodes de stockage de données dans notre algorithme. Nous donnons par la suite deux exemple d'utilisation de l'interface AXI Stream et BRAM.

— Pour le transfert des données, les ports d'entrée/sortie sont configurés pour utiliser le protocole AXI-Stream avec des signaux de communication minimum (DATA, VALID et READY). Le bloc de la multiplication modulaire CIOS qui est généré avec Vivado HLS sera connecté par les interfaces AXI-Stream au contrôleur DMA(Direct Memory Access). La génération d'adresses et la planification de la transaction mémoire sont gérées par l'AXI DMA IP.

— Dans le cas où le stockage de données se fait dans la BRAM, nous utilisons le contrôleur de BRAM afin d'accéder aux adresses de la mémoire.

B) **Directive UNROLL** est utilisé pour appliquer le parallélisme entre les itérations des boucles. En fait, les opérandes des entrées a, b sont stockés dans la mémoire BRAM de l'FPGA, et la mémoire BRAM est caractérisée

par des ports doubles. Par la suite, nous avons configuré les ports doubles comme suit : un port pour chaque opération, deux ports d'écriture ou deux ports de lecture.

C) **Directive PIPELINE** Dans notre conception, les itérations de la boucle sont pipelinées avec une seule différence de cycle d'horloge entre les deux en appliquant Loop pipelining avec une Initiation Interval (II) = 1 qui correspond au nombre de cycles d'horloge entre les instants de début des itérations de boucle consécutives afin de réduire le temps de latence d'exécution.

2.3 Évaluation des performances de l'intégration de l'IP matérielle

L'étape finale, et après avoir fixé l'adresse et après la validation de notre conception, nous avons vérifié le résultat de l'IP matérielle avec notre solution logicielle avec la plate-forme (SDK : Software Development Kit) 2016.4, qui devrait répondre aux mêmes résultats. Tout d'abord, les données en entrée sont stockées dans la mémoire BRAM. Ces données sont chargées au IP (Algorithme Montgomerie CIOS) en utilisant la communication AXI-DMA à travers le processeur (ZYNQ). Par la suite, on lance l'outil SDK afin de calculer le temps d'exécution de l'IP en utilisant la bibliothèque *xtime_l.h*. Une fois que la multiplication modulaire est terminée, le résultat est renvoyé à la mémoire BRAM à travers le chemin inverse. Afin de vérifier l'exactitude de notre résultat, nous avons comparé les résultats donnés par le FPGA avec le code python.

2.3.1 Analyse des Résultats

Nous commençons par comparer les résultats des différents algorithmes de réduction modulaire (CIOS, AMNS) pour différentes tailles d'entiers premiers. Les conceptions ont été décrites dans une synthèse de haut niveau (HLS) et synthétisées pour la Zedboard Zync-7000 XC7z020 des FPGA Xilinx. Le tableau 2.1 montre les résultats pour les combinaisons de directives d'optimisation que nous avons retenus.

La conception de l'AMNS offre la possibilité d'être utilisée dans diverses applications avec différentes longueurs de bits comme ECC, HECC. Comme le montre le tableau 2.1, nous avons un résultat intéressant pour notre conception

	CIOS	AMNS					
p size bit	128	128			256		512
w **bit**	**32**	**32**	**36**	**54**	**36**	**64**	**64**
Freq (Mhz)	115.6	114.8	118.06	115.6	118.06	118.063	118.06
Cycle	246	497	267	193	926	457	1637
Vitesse μs	2.12	4.32	2.26	1.66	7.84	3.87	13.86
Slice DSP	60	18	51	106	48	96	96
Slice LUT	5681	3048	3617	4799	2585	3576	4250
BRAM	0	2	10	11	12	16	20

TABLE 2.1 – Comparaison de nos architectures.

AMNS dans le fait que les nombres de cycles sont indépendants de la taille des bits du mot. Nous avons également mis en œuvre la version CIOS de la multiplication de Montgomery afin d'effectuer une comparaison équitable. En effet, nos implantations sont homogènes et si l'implantation la plus efficace de l'algorithme CIOS existe, cela implique que la multiplication AMNS pourrait également être améliorée.

Nous avons pu constater que, compte tenu de la taille des mots qui est indiquée ici w, les résultats AMNS sont meilleurs que les résultats CIOS, en considérant chaque point de comparaison ; le nombre de cycles d'horloge et le nombre de ressource. Si l'on considère la fréquence utilisée, pour $p = 128$, et si on compare les résultats du CIOS et du système AMNS avec une taille des mots de 32 bits, on constate que le système AMNS nécessite moins de DSP et d'horloge que le système CIOS.

2.4 Conclusion

Dans ce chapitre, nous avons construit deux conceptions de matériel IP avec la synthèse de haut niveau (HLS), telles que la méthode CIOS de l'algorithme d'application de Montgomery et le système AMNS (Adapted Modular Number System) sur de grands corps finis caractéristiques premiers $\mathbb{F}_p$. Notre motivation était de fournir une comparaison équitable entre l'implémentation matérielle de la multiplication AMNS et la multiplication de Montgomery classique. Nous donnons les résultats de nos deux conceptions IP après routage et placement à l'aide d'un FPGA Xilinx Zync-7000 XC7z020 de Zedboard. Notre architec-

ture peut être utilisée avec différents niveaux de sécurité : 128, 256 ou 512. Nous avons mis en œuvre la première conception de multiplication modulaire AMNS, dans le but d'optimiser la conception avec une architecture de synthèse haut niveau. Notre conclusion est que es résultats obtenu de l'implantation de la multiplication modulaire en AMNS est compétitif à celle de Montgomery CIOS d'une part. D'autre part, les performances de l'implantation changes en changeant de langage de programmation. Pour cela, afin d'améliorer les performances de notre conception AMNS, qui est la première implantation faite pour AMNS dans le FPGA, il pourrait être intéressant de fournir une implantation en VHDL.

3 Conception et implantation RTL de la multiplication en AMNS

Sommaire

Dans le chapitre 1, nous avons détaillé les différents types de multiplications en mentionnant quelques algorithmes. Notre objectif est la conception et l'implémentation de la multiplication modulaire dans le système de représentation AMNS (Adapted modular number system). En raison de sa régularité et de sa simplicité, cet algorithme peut être facilement implémenté dans les DSPs des FPGAs.

La plupart des solutions proposées dans la littérature utilisent des algorithmes complexes et coûteux pour améliorer les performances de calcul dans leurs implémentations. En raison de la complexité de ces algorithmes, les multiplications modulaires rapides consomment souvent une quantité importante de ressources matérielles des FPGAs. Dans le chapitre précédent nous avons présenté la première implantation qui est décrite avec l'outil Vivado HLS. Alors, dans ce chapitre nous avons décidé de décrire notre architecture en VHDL où on a plus la main sur l'architecture que nous proposons.

Notre objectif est de construire la première architecture matérielle de la multiplication modulaire en AMNS. Pour cela, nous décomposons les éléments de taille p dans $\mathbb{F}_p$ en un ensemble de mots de w bit de sorte que w soit un multiple de 18. Le choix de multiple de 18 est justifié afin de mieux utiliser les DSP 48 du FPGA, puisque les blocs de DSP sont des unités de 18×25 bits. Ce choix permet de réduire le nombre de blocs de DSP utilisés dans l'implémentation. Nous indiquons n le nombre de coefficients de w bit nécessaires pour chaque polynôme avec $p = nw$.

Le travail que nous présentons dans ce chapitre a été publié dans l'article de revue [CDD+21].

3.1 Modélisation d'une nouvelle architecture

Dans cette section de ce chapitre, on va s'intéresser à l'implémentation matérielle de l'architecture de la multiplication modulaire dans le système AMNS. Cette multiplication est basée sur l'algorithme 12.

Les étapes de cet algorithme sont développés comme suit :

Algorithm 12 Multiplication modulaire en AMNS - les étapes principales

$A, B \in \mathcal{B}$ et $M, M' \in \mathbb{Z}_n[X]$

$S \in \mathcal{B}$

$$C \leftarrow A \times B \tag{3.1}$$

$$C' \leftarrow C \bmod E \tag{3.2}$$

$$Q \leftarrow C' \times M' \bmod (E, 2^r) \tag{3.3}$$

$$R \leftarrow ((Q \times M)) \bmod E \tag{3.4}$$

$$S \leftarrow (C' + R)/2^r \tag{3.5}$$

Nous rappelons que les entrées A, B et la sortie C sont des polynômes de degré $n-1$. Le polynôme M est le polynôme de réduction interne et l'inverse modulaire de son opposé modulo (E, ϕ) est M'. Sachant que le polynôme de réduction externe E est un polynôme de la forme $E(X) = X^n - \lambda$. Notez, pour des raisons d'efficacité, nous choisissons $\phi = 2^r$. Les polynômes C, C' et R sont des polynômes intermédiaires. Cet algorithme produit le polynôme de sortie S étant le résultat de la multiplication de A par $B \bmod p$. Les coefficients des polynômes A, B, C et les constantes M et M' sont stockés dans la mémoire BRAM, ainsi le résultat est réécrit dans la même mémoire.

La modélisation de notre architecture est illustrée par la figure 3.1. Cette architecture est composé par trois modules :

— Mémoires : Afin de mémoriser les résultats intermédiaires.

— AMNS Multiple-Coefficient : Afin de faire la multiplication modulaire en AMNS.

— Controleur FSM : Qui va commander

Dans cette structure, l'opérateur *AMNS Multiple-Coefficient*, traite les étapes (3.1), (3.2), (3.3), (3.4) et (3.5) de l'algorithme de multiplication AMNS. En premier lieu, il charge le coefficient de chaque valeur d'entrée A, B, M' et M de la mémoire. Ensuite, il traite les résultats intermédiaires C', Q, R, S et les stocke dans des registres locaux dédiés. Notamment, cela permet d'utiliser C', Q et R dans l'opérateur *AMNS Multiple-Coefficient* comme valeurs intermédiaires pour les entrées dans le cycle qui suit. Ainsi, le registre dédié à S fournit le résultat à la mémoire.

En outre, les plateformes cibles ont rarement assez de broches d'entrée pour traiter des données de taille cryptographique en un seul cycle. Par exemple, la carte utilisée pour nos implantations n'a que 220 broches d'entrée/sortie alors que le chemin de données de multiplication modulaire est conçu jusqu'à 1024 bits.

Pour utiliser efficacement les ressources du FPGA, nous avons choisi une architecture dans laquelle la taille w des mots est un multiple de 18 bits. Les architectures que nous détaillons dans la partie qui suit sont configurés avec des tailles de coefficients de 36 bits. Par la suite, nous détaillons les différentes étapes traitées dans l'unité *AMNS Multiple-Coefficient* pour un exemple de $w = 36$ bits et un module p de 128 bits.

Pour augmenter les performances de l'implantation, nous avons combiné la multiplication polynomiale (3.1) et la réduction polynomiale (3.2). En effet, le polynôme de réduction externe $E = X^n - \lambda$ est un polynôme dont la propriété

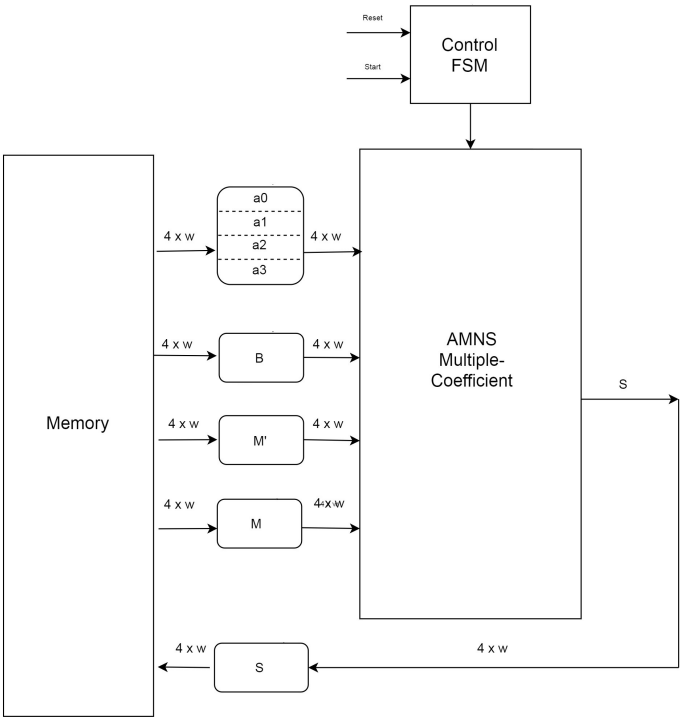

FIGURE 3.1 – Architecture du système

est $X^n \bmod E = \lambda$. Ainsi, le produit $C = A \times B$ et la réduction $C' = C \bmod E$ peuvent être combinés. Nous pouvons directement calculer $C' = A \times B \bmod E$ comme suit :

— $c'_0 = a_0b_0 + \lambda(a_1b_3 + a_2b_2 + a_3b_1)$,

— $c'_1 = a_0b_1 + a_1b_0 + \lambda(a_2b_3 + a_3b_2)$,

— $c'_2 = a_0b_2 + a_1b_1 + a_2b_0 + \lambda(a_3b_3)$,

— $c'_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$.

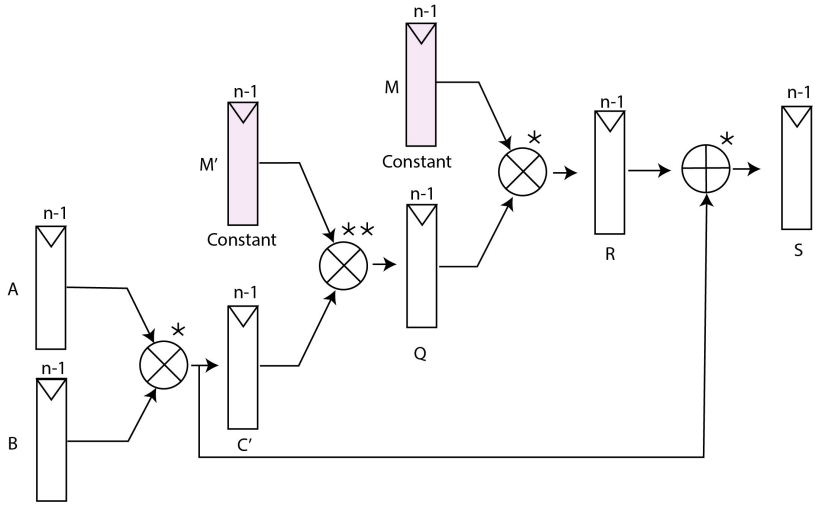

FIGURE 3.2 – Illustration des opérations de multiplication modulaire dans le système AMNS, $(n-1)$ est le degré du polynôme. La notation "*" représente $\bmod(E)$, et "**" représente $\bmod(E,\phi)$

La multiplication polynomiale est généralement implémentée à l'aide des multiplicateur-accumulateur (opérateurs MAC) avec une largeur de données d'opération fixe (par exemple, 36 bits). La deuxième étape consiste à calculer $Q = C' \times M' \bmod (E, 2^r)$ comme suit :

— $Q_0 = c'_0m'_0 + c'_1m'_3 + c'_2m'_2 + c'_3m'_1$,

— $Q_1 = c'_0m'_1 + c'_1m'_0 + c'_2m'_3 + c'_3m_2$,

— $Q_2 = c'_0m'_2 + c'_1m'_1 + c'_2m'_0 + c'_3m_3$,

— $Q_3 = c'_0m'_3 + c'_1m'_2 + c'_2m'_1 + c'_3m'_0$.

La troisième étape consiste à calculer $R = ((Q \times M)) \bmod E$:

— $R_0 = q_0m_0 + q_1m_3 + q_2m_2 + q_3m_1$,

— $R_1 = q_0m_1 + q_1m_0 + q_2m_3 + q_3m_2$,

— $R_2 = q_0m_2 + q_1m_1 + q_2m_0 + q_3m_3$,

— $R_3 = q_0m_3 + q_1m_2 + q_2m_1 + q_3m_0$.

La dernière étape est simplement $S = (C' + R)/2^r$. La multiplication modulaire est donc traitée à travers quatre étapes, de sorte que le résultat final S est disponible au cinquième état. Ceci est illustré dans la figure 3.2.

3.2 Implantation Matérielle

Dans cette partie, nous proposons deux architectures qui peuvent être utilisées pour calculer une multiplication modulaire dans le système AMNS sur FPGA. Afin de choisir la plus performante. La première est une architecture séquentielle, qui est facile à réaliser car les opérations arithmétiques sont effectuées dans l'ordre décrit par la figure 3. L'architecture semi-parallèle vise à exécuter le plus grand nombre d'opérations arithmétiques dans le même cycle pour avoir une conception rapide. Une analyse préliminaire montre que l'architecture séquentielle utilise moins de registres que l'architecture semi-pipeline, mais nécessite plus de cycles d'horloge pour effectuer la multiplication modulaire dans le système AMNS.
Dans la partie 3.2.1, nous détaillons l'architecture séquentielle. Ensuite, nous présentons notre architecture semi-parallèle dans la partie 3.2.2.

3.2.1 Architecture séquentielle

Dans cette architecture, les étapes décrites dans la partie 3.1 sont programmées comme indiqué dans la figure 3.3. En effet, cette figure est un exemple d'ordonnancement d'une multiplication modulaire dans AMNS, dont les entrées A et B ont quatre coefficients de 36 bits et le module p est un entier de 128 bits.

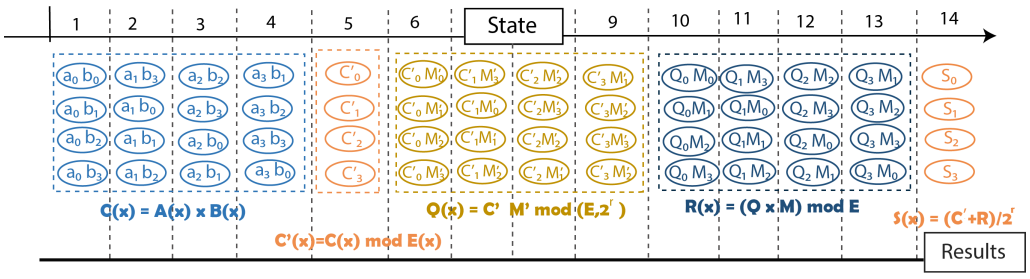

FIGURE 3.3 – Architecture séquentielle proposée, pour $n = 4$, $w = 36$ et la taille de $p = 128$

Dans la figure 3.1 nous implémentons le bloc *AMNS Multiple-Coefficient* qui exécute l'algorithme précédent comme suit.

D'abord, il prend en entrée les variables de boucle i et j du contrôleur et les fournit à la mémoire en tant qu'entrée d'adresse. Les variables d'entrée a_i, b_j, M'_j, et M_j sont des tableaux de coefficients de w bits. Ici, $w = 36$ pour a_i, b_j, $w = 39$ pour M'_j et $w = 40$ pour M_j.

Le bloc *AMNS Multiple-Coefficient* calcule les 5 étapes de l'algorithme 12. En effet, en premier lieu l'étape (3.1) calcule une variable intermédiaire C étape

par étape, du coefficient le moins significatif au coefficient le plus significatif. Par conséquent, à chaque cycle une nouvelle valeur de C_0, C_1, C_2 et C_3 apparaît (Fig. 3.2).

Ensuite, le processus exécute l'étape (3.2) afin de calculer une autre variable intermédiaire C' de manière similaire à l'étape (3.1). En fait, le polynôme C reçoit d'abord le coefficient le moins significatif en série afin de calculer le coefficient le moins significatif de C'.
En outre, les étapes (3.3) et (3.4) qui calculent les variables intermédiaires Q et R sont effectuées de la même manière.

Finalement, la division à l'étape (3.5) est effectuée par un simple décalage à droite de r bits. Le résultat final est calculé à cette étape.

Les étapes (3.1), (3.3) et (3.4) nécessitent n itérations pour être calculées, tandis que les étapes (3.2) et (3.5) sont effectuées dans une seule étape. Les figures 3.4, 3.5, 3.6 et 3.7 montrent les parties du bloc *AMNS Multiple-Coefficient* qui calculent les étapes (3.1), (3.2), (3.3), (3.4) et (3.5). Bien que ces chiffres ne le montrent pas, l'architecture de multiplication AMNS comprend une mémoire à port unique d'une largeur de 36 bits, qui permet l'accès en lecture ou en écriture à une seule adresse en un cycle d'horloge. Cette mémoire stocke les coefficients d'entrée A, B, M et M' pour le calcul et l'enregistrement de la valeur de sortie S. Elle sert également de stockage temporaire pour les variables intermédiaires C', R et Q.

Nous décrivons dans les sous-sections qui suivent les architectures matérielles de toutes les étapes de l'architecture séquentielle.

3.2.1.1 Architecture matérielle des étapes (1) et (2)

L'architecture matérielle du module qui calcule les étapes (1) et (2) est illustré dans la figure 3.4.

Chaque coefficient de C', qui ont chacune une taille 72 bits, est calculé avec un opérateur MAC d'entrée de 36 bits. Par exemple, pour une architecture de module de 128 bits, quatre opérateurs MAC sont nécessaires à cette étape.

En fait, un tel chemin de données nécessite 3 blocs DSP pour chaque opérateur MAC. Chaque MAC effectue une multiplication des coefficients de A et B pris en mémoire et accumule le résultat. La valeur finale est stockée dans les registres C'_i qui peuvent produire le résultat en série. Il convient de noter que l'étape (3) qui a utilisé C' doit être calculée modulo(E,ϕ). Ainsi, dans cet exemple, seuls 40 bits des coefficients c'_i sont nécessaires à l'étape suivante. En effet, c'est l'application du $\mod \phi$

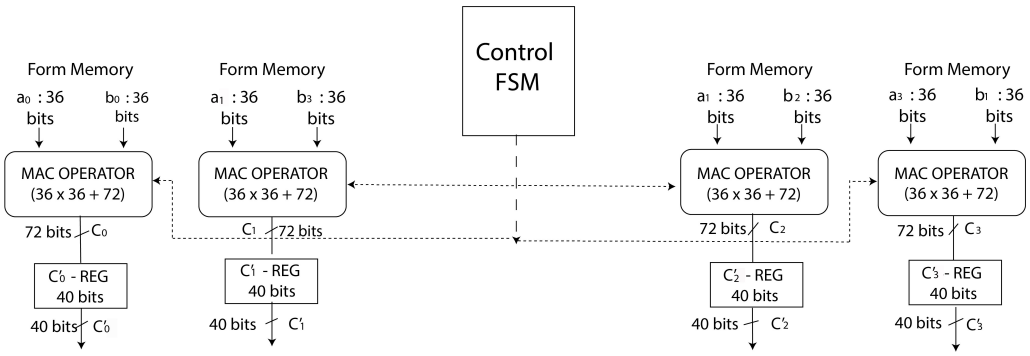

FIGURE 3.4 – Architecture séquentielle, calcul de l'étape (1) et (2) pour $n = 4$, $w = 36$ et $p = 128$

3.2.1.2 Architecture de l'étape (3)

L'architecture du module qui calcule l'étape (3) de la multiplication modulaire est similaire à celle du module précédent, à l'exception de la taille des entrées et des sorties (figure 3.5). Dans ce module, chaque opérateur MAC prend en entrée un coefficient de 39 bits de M' et un coefficient de 40 bits de C'. Chaque opérateur MAC sort un coefficient de 79 bits de Q, mais seuls 40 bits sont conservés car l'étape (3) est calculée par le modulo (E,ϕ).

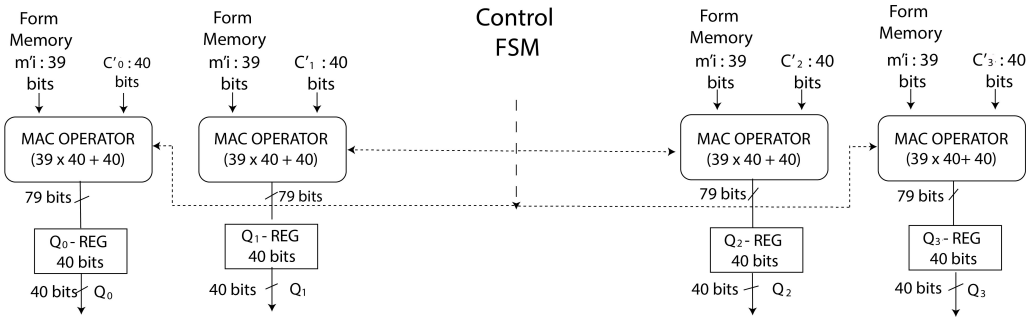

FIGURE 3.5 – Architecture séquentielle, calcul de l'étape (3) avec $p = 128$ bits et $w = 36$ bits

3.2.1.3 Architecture de l'étape (4)

L'architecture matérielle du module qui calcule l'étape (4) est illustré par la figure 3.6. Ce module est similaire au module présenté dans la figure 3.4, à l'exception de la taille des entrées et de la sortie. Chaque opérateur MAC reçoit un coefficient de 36 bits de M et un coefficient de Q qui est donné en série. Chaque opérateur produit un coefficient de 76 bits de R qui est stocké dans un registre de 76 bits (non affiché dans la figure 3.6).

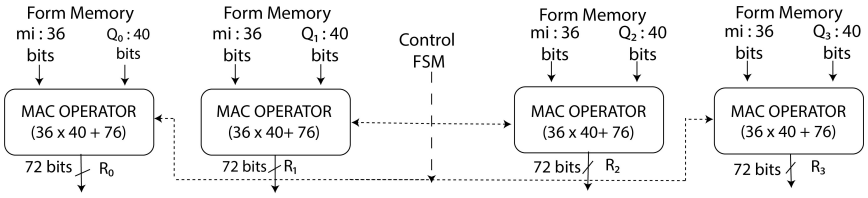

FIGURE 3.6 – Architecture séquentielle, calcul de l'étape (4) avec $p = 128$ bits et $w = 36$ bits

3.2.1.4 Architecture de l'étape (5)

La dernière étape (5) consiste en une somme sur 76 bits et un décalage. L'architecture matérielle qui décrit le module qui calcule cette étape est illustré dans la figure 3.7. Chaque additionneur introduit un coefficient de 72 bits de C' et un coefficient de 76 bits de R. L'opérateur de décalage suivant correspond à la division de l'étape (5). Dans cet exemple, le diviseur est 2^{40}. Les coefficients de 36 bits du résultat S sont remis en mémoire.

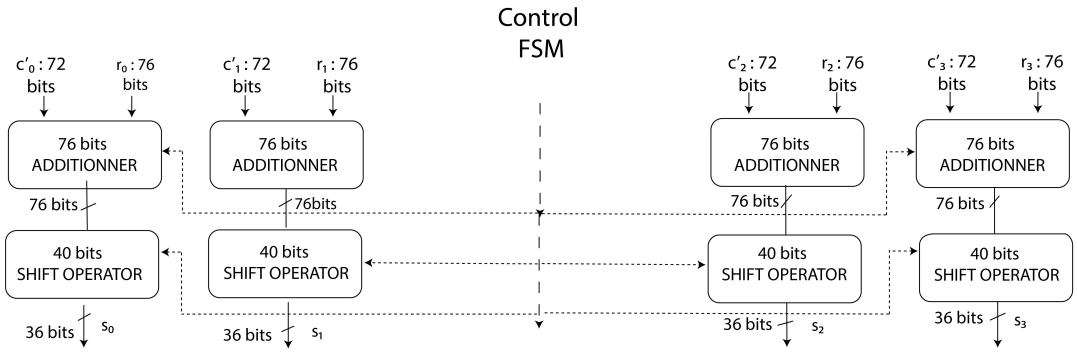

FIGURE 3.7 – Architecture séquentielle, calcul de l'étape (5) avec $p = 128$ bits et $w = 36$ bits

3.2.1.5 Le contrôleur FSM de l'architecture séquentielle

Les signaux d'interface du contrôleur sont réglés en fonction de la machine à états finis (FSM : Finite-State Machine). Cette FSM est illustrée par la Fig 3.8 pour un module p de taille 128 bits, où l'état $\mathcal{S}_0$ est l'état initial. Chaque état de notre FSM nécessite l'exécution de deux cycles, sauf l'état final qui est exécuté en un seul cycle. Le contrôleur a 14 états. Notre architecture séquentielle de

base nécessite un total de 28 cycles pour calculer une multiplication modulaire pour un module p de 128 bits et des coefficients de taille 36 bits.

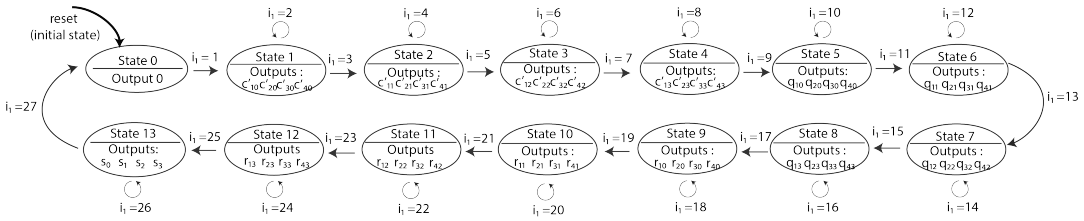

FIGURE 3.8 – Machine à état finis pour $w = 36$ et avec une taille de 128 bits du module p - Architecture séquentielle

Le cheminement des données se compose de huit registres internes, d'un compteur et d'un comparateur. Le contrôleur reste dans l'état $\mathcal{S}_0$ jusqu'à ce que l'instruction START soit définie. Lorsque le signal START est défini, les registres a_i et b_i sont chargés avec des valeurs d'entrée, les registres $C_{00}, C_{10}, C_{20}, C_{30}$ et le compteur sont réinitialisés. Les différents états sont résumés dans le tableau 3.1. Les autres états fonctionnent comme suit :

— Dans l'état $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ et $\mathcal{S}_4$ la machine FSM exécute les étapes (3.1) et (3.2) de l'algorithme.

— $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ et $\mathcal{S}_4$ attendent de valider le signal $DONE\_C'$. Une fois que les coefficients du polynôme C' sont chargés dans le registre BRAM, la FSM passe à l'état suivant.

— Les états $\mathcal{S}_5$, $\mathcal{S}_6$, $\mathcal{S}_7$ et $\mathcal{S}_8$ sont en attente pour valider le signal $DONE\_Q$. Ils calculent l'étape (3.3) de l'algorithme. Une fois que les coefficients du polynôme Q sont chargés dans le registre BRAM, le FSM passe à l'état suivant.

— Les éléments $\mathcal{S}_9$, $\mathcal{S}_{10}$, $\mathcal{S}_{11}$ et $\mathcal{S}_{12}$ contrôlent le calcul de l'étape (3.4) de l'algorithme. Ils attendent de valider le signal $DONE\_R$. Une fois que la machine FSM génère les coefficients du polynôme R et les charge dans le registre BRAM, la FSM passe à l'état suivant.

— L'état $\mathcal{S}_{13}$ attend de valider le signal $DONE\_S$. Il calcule la dernière étape de l'algorithme.

états	Transition	Condition
Initialisation	$\mathcal{S}_0$	reset = done
États (1) et (2)	$\mathcal{S}_0 \rightarrow \mathcal{S}_1$ $\mathcal{S}_1 \rightarrow \mathcal{S}_2$ $\mathcal{S}_2 \rightarrow \mathcal{S}_3$ $\mathcal{S}_3 \rightarrow \mathcal{S}_4$	$DONE\_C = 1$ $DONE\_C' = 1$
État (3)	$\mathcal{S}_4 \rightarrow \mathcal{S}_5$ $\mathcal{S}_5 \rightarrow \mathcal{S}_6$ $\mathcal{S}_6 \rightarrow \mathcal{S}_7$ $\mathcal{S}_7 \rightarrow \mathcal{S}_8$	$DONE\_Q = 1$
État (4)	$\mathcal{S}_8 \rightarrow \mathcal{S}_9$ $\mathcal{S}_9 \rightarrow \mathcal{S}_{10}$ $\mathcal{S}_{10} \rightarrow \mathcal{S}_{11}$ $\mathcal{S}_{11} \rightarrow \mathcal{S}_{12}$	$DONE\_R = 1$
État (5)	$\mathcal{S}_{12} \rightarrow \mathcal{S}_{13}$	$DONE\_S = 1$

TABLE 3.1 – Tableau de transition des étapes de la FSM

3.2.2 Architecture semi-parallèle

Nous présentons dans cette partie notre architecture semi-parallèle qui vise à exécuter le maximum d'opérations intermédiaires en parallèle afin de minimiser le nombre de cycles. Notre architecture n'est pas complètement parallèle, car le passage de l'étape (3.3) à l'étape (3.4) doit se faire séquentiellement. A titre d'exemple, nous décrivons dans la figure 3.9 un ordonnancement des étapes de l'algorithme 12, avec $n = 4$ et $w = 36$. Il apparaît qu'il n'est pas nécessaire

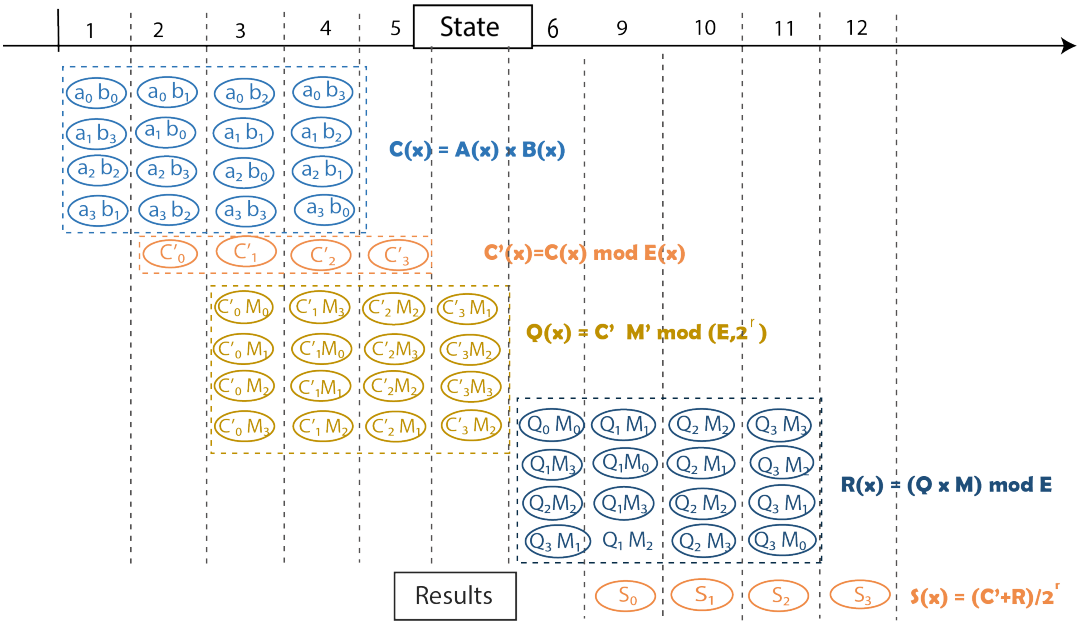

FIGURE 3.9 – Ordonnancement de l'architecture semi-parallèle

d'attendre la fin du calcul du dernier coefficient de C à l'étape (3.1) avant de commencer le calcul de l'étape (3.2) (Alg.12). En effet, à l'étape 1, nous

calculons en parallèle les termes qui doivent être ajoutés pour obtenir c'_0. Ainsi, le calcul des coefficients C'_0 commence à l'étape 2 (Fig. 3.9). Il est possible d'observer la même chose avec les étapes (2) et (3) ainsi que les étapes (4) et (5).

En conséquence, notre architecture semi-parallèle est composée de 4 modules comme détaillé dans les figures 3.10, 3.11 3.12 et 3.13. Chacun d'entre eux, prend en charge une ou deux étapes de l'algorithme 12. Ces modules sont :

— Le module 1 (Fig. 3.10) : La multiplication $a_i \times b_j$ se fait par la cellule $MULTIPLIER$. La cellule $ADDER$ collecte les produits et calcule c'_i, qui sont les coefficients du polynôme C.

— Le module 2 (Fig. 3.11) : La multiplication $c'_i \times m'_i \mod (\phi)$ génère les coefficients du polynôme Q.

— Le module 3 (Fig. 3.12) : La multiplication des coefficients de Q par M mod E génère R.

— Le module 4 (Fig. 3.13) : L'addition $C' + R$, suivie de la division exacte par 2^r, génère les coefficients de la sortie S.

3.2.2.1 Architecture matérielle du module 1

Le premier module de la figure 3.10 est chargé des étapes (3.1) et (3.2).

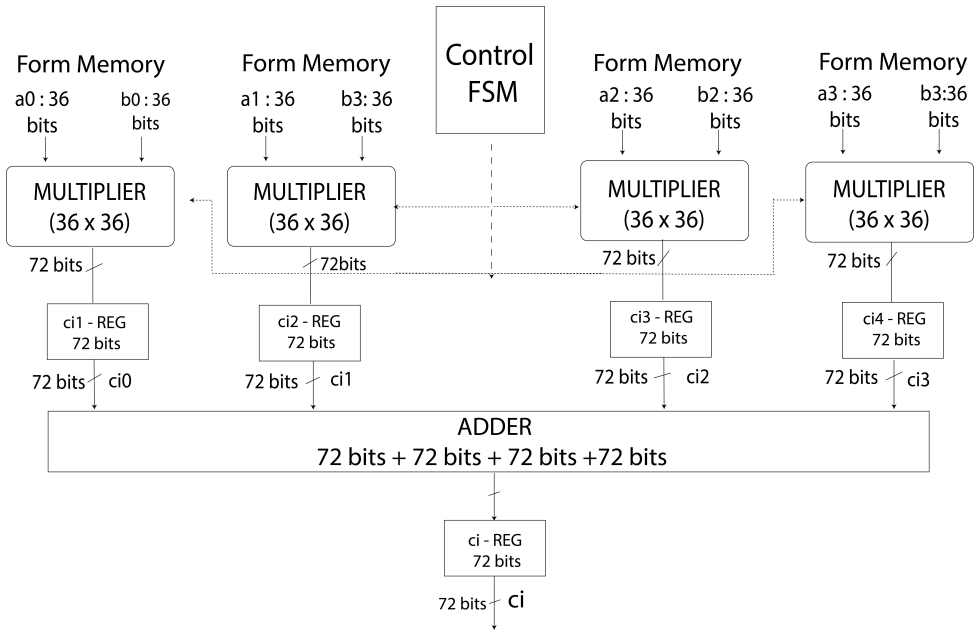

FIGURE 3.10 – Architecture semi-parallèle, calcul des étapes (1) et (2) avec $w = 36$ et un module p de 128 bits

La première partie de ce module est composée de multiplicateurs qui calculent le produit des coefficients 36 bits de A et B en un cycle. Le coefficient de 72 bits qui en résulte est stocké dans les registres.

La deuxième partie de ce module est composée d'un opérateur Adder qui collecte les produits intermédiaires et les additionne afin de calculer un coefficient de C. Dans cet exemple $C_i = C_{i0} + C_{i1} + C_{i2} + C_{i3}$. La sortie de l'additionneur est connectée à un démultiplexeur qui fournit le coefficient dès que son calcul est terminé. Dans cet exemple, ce module a besoin de 5 cycles pour terminer le calcul. Il faut noter que chaque état est exécuté en deux cycles d'horloge.

3.2.2.2 Architecture matérielle du module 2

Le module de la figure 3.11 est conçu pour calculer l'étape (3.3) de l'algorithme 12. Il est composé d'opérateurs MAC qui prend en entrée un coefficient de 39 bits de M' et un coefficient de C'. Ce coefficient est acquis en série. Chaque opérateur produit un coefficient de Q qui est réécrit en série dans la mémoire.

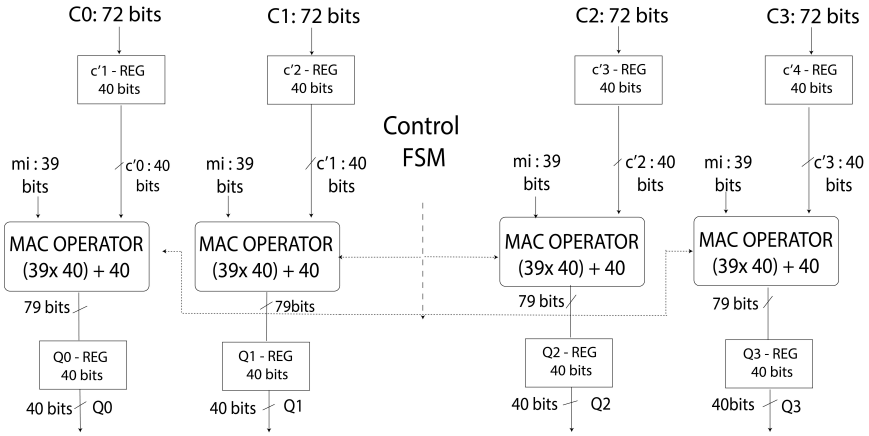

FIGURE 3.11 – Architecture semi-parallèle, calcul de l'étape (3) avec $w = 36$ et de module p de 128 bits

3.2.2.3 Architecture matérielle du module 3

L'opérateur de module de la figure 3.12 calcule l'étape (4). Il est composé de multiplicateurs qui reçoivent les coefficients de M et Q. Les multiplieurs sur des entiers ont deux entrées une de 40 bits et une sur 36 bits et produisent des résultats de 76 bits. Le résultat est stocké dans les registres $R_i - Reg$ qui sont entrés dans l'additionneur qui calcule un coefficient de 76 bits de R.

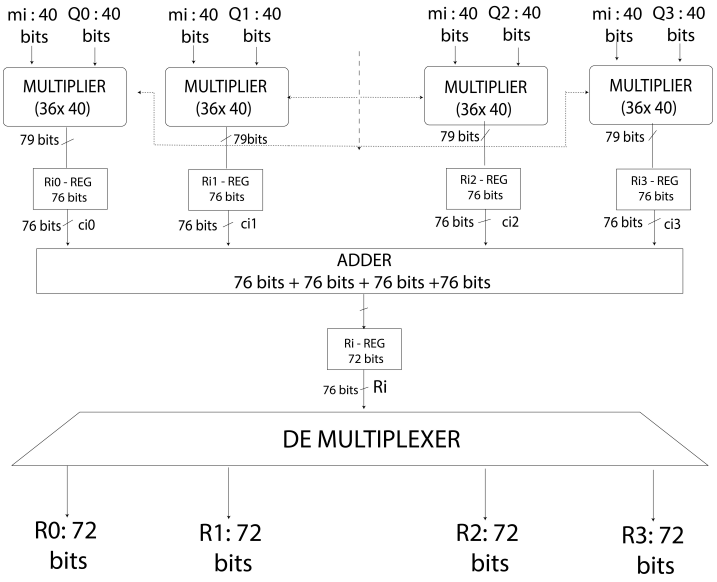

FIGURE 3.12 – Architecture semi-parallèle, calcul de l'étape (4) avec $w = 36$ bits et a 128 bits la valeur du module de p

La sortie est connectée à un démultiplexeur qui produit tous les coefficients calculés de R.

3.2.2.4 Architecture matérielle du module 4

Le dernier module (Fig. 3.13) calcule la dernière étape de la multiplication modulaire dans le système de représentation AMNS.

Il est composé d'additionneur 76 bits et prend en entrée les coefficients de 72 bits de C' et les coefficients de 76 bits de R. Dans cette étape, une division par 2^r est effectuée. Ainsi, seuls les 36 bits les plus significatifs de la somme sont produits.

3.2.2.5 Main controller (MC)

Cette conception est contrôlée par une machine à états finis (FSM). Pour assurer la synchronisation entre les modules de l'architecture. La figure 3.14, représente les états de la FSM qui contrôle l'architecture semi-parallèle, pour $w = 36$ bits et une taille du module p égale à 128 bits. La FSM est composée de 11 états, l'état $\mathcal{S}_0$ étant l'état de départ.

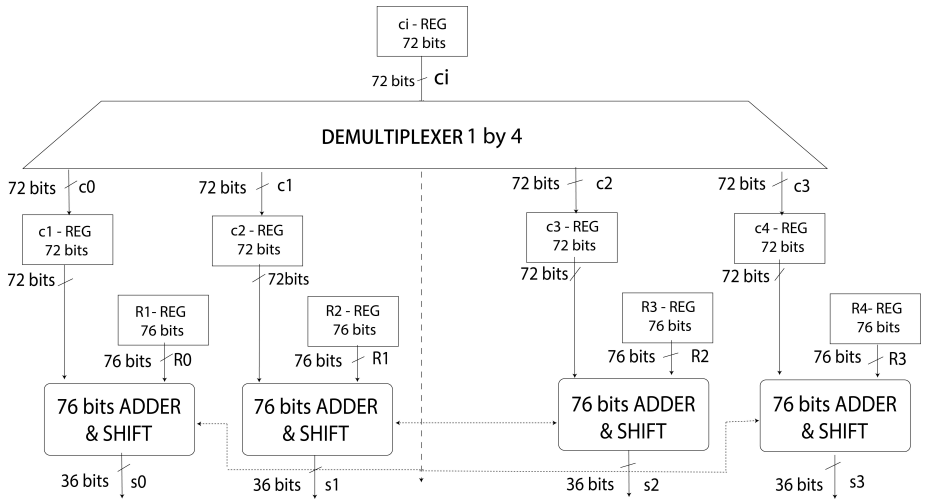

FIGURE 3.13 – Architecture semi-parallel, calcul de l'étape (5) avec $w = 36$ et a 128-bit la valeur de p

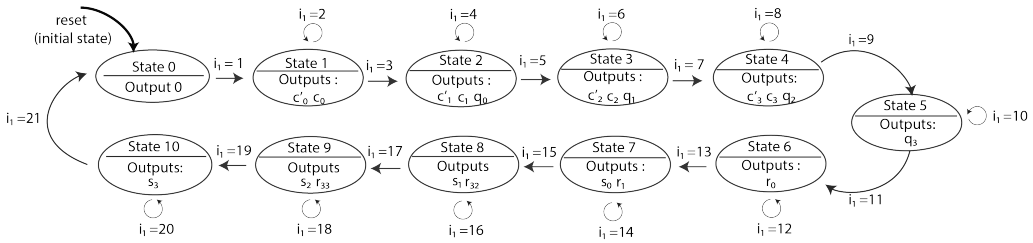

FIGURE 3.14 – Machine à état finis pour l'architecture semi-parallèle avec $w = 36$ et $p = 128$ bits

Dans cette FSM, chaque état nécessite deux cycles pour son exécution, sauf l'état final qui est exécuté en un seul cycle. Cela nous donne un coût total de 21 cycles. Le contrôleur reste dans l'état $\mathcal{S}_0$ jusqu'à ce que le signal START soit activé.

Une fois que le signal START est activé, les entrées a_i et b_i sont chargées, les registres $c_{00}, c_{01}, c_{02}, c_{03}$ et le compteur sont réinitialisés.

Le passage de la FSM d'un état à l'autre, est réalisé lorsque la transition entre deux états soit valide.

La fin de la FSM est signalé par le changement d'état du bit $DONE$ de 0 à 1.

— Le premier état est un état initial $\mathcal{S}_0$. Pendant cet état, les coefficients a_i et b_i sont chargés dans les registres.

— Une fois que les états $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ et $\mathcal{S}_4$ sont terminés, le signal $DONE\_C'$

est activé. Ces états permettent de calculer les coefficients du polynôme C' et de les charger dans le registre BRAM. Cela correspond aux étapes (1) et (2).

— Dans l'état $\mathcal{S}_1$, la machine FSM utilise un opérateur MAC pour exécuter les équations (1) et (2) de l'algorithme 12. Dans cet état, la machine calcule le coefficient c_0 du polynôme C'.

— Dans l'état $\mathcal{S}_2$, la machine calcule le deuxième coefficient c'_1 de C'. En parallèle, elle déclenche le calcul du coefficient q_0 du polynôme Q. Une fois que les valeurs des coefficients du polynôme q_0 et c'_1 sont chargées dans le registre BRAM, la FSM passe à l'état suivant.

— Les états $\mathcal{S}_3$, $\mathcal{S}_4$ et $\mathcal{S}_5$, visent à valider les signaux $DONE\_q_1$, $DONE\_q_2$, $DONE\_q_3$, $DONE\_c'_2$, $DONE\_c'_3$. La FSM utilisent huit opérateurs MAC dans chaque État pour exécuter l'étape (3). Une fois que tous les coefficients du polynôme Q sont générés, la FSM passe à l'état suivant.

— L'état $\mathcal{S}_6$, valide le signal $DONE\_r_0$. Il le calcule les opérations de l'étape (4),

— Les états $\mathcal{S}_7$, $\mathcal{S}_8$, $\mathcal{S}_9$, vise à valider les signaux $DONE\_r_1$, $DONE\_r_2$, $DONE\_r_3$, $DONE\_s_0$, $DONE\_s_3$ et $DONE\_s_2$. Pendant ces états, les coefficients de $Q \times M$ sont calculés en parallèle. Cela correspond à l'étape (5).

— L'état $\mathcal{S}_{10}$, définit le signal $DONE\_s_3$. Il calcule le quatrième coefficient du polynôme S en utilisant un opérateur d'addition et un opérateur de décalage.

3.3 Résultats expérimentaux

Dans cette partie nous présentons les résultats d'implantations des architectures semi-parallèle et séquentielle sur FPGA. Nous avons choisi deux familles différentes de FPGA, afin de confirmer nos résultats avec deux technologies différentes. Nos conceptions ont été décrites en langage VHDL, synthétisées, placées et routées avec les outils ISE 14.7 pour les FPGA Xilinx et Quartus II 12.1 pour le FPGA Intel-Altera en utilisant les options de synthèse par défaut.

Les résultats sont donnés pour 4 plateformes FPGA, deux plateformes de hautes performances et deux plateformes de basse performances :

— Deux FPGA à faible coût : Aria II GX (EP2A6x45DF2915) et Zynq xc7 (z010-3g400),

— Deux FPGA haute performance : Virtex 6 (X6vlx 75 t-3ff484) et Cyclone V (5cgXFC7DF 31 C8ES)

Nous présentons ci-dessous les résultats d'une multiplication modulaire avec différentes tailles de coefficient w et pour différentes tailles de p : 128, 256, et 512 bits pour les FPGA qui sont suffisamment grands. Nous indiquons ici la fréquence de fonctionnement maximale des architectures, le nombre de cycles nécessaires pour réaliser la multiplication modulaire, la durée totale du calcul, le nombre de blocs DSP et de LUT nécessaires à l'implémentation.

Famille	Xilinx				
FPGA	Zynq		Virtex 6		
p bit	128	256	128	256	512
w	36		36		72
Fq(Mhz)	256.5	242.3	238	225.8	191
Cycles	19	33	19	33	33
Temps μs	0.074	0.136	0.079	0.146	0.172
DSP	60	56	60	120	236
LUT	1025	12105	1035	2738	18964

TABLE 3.2 – Résultats des implémentations de l'architecture semi-parallèle sur la famille FPGA de Xilinx

Famille	Intel - Altera				
FPGA	Aria II		Cyclone V		
p bit	128	256	128	256	512
w	36		36		72
Fq(Mhz)	218.2	194.4	157.1	106.4	101.6
Cycles	19	33	19	33	33
Temps μs	0.087	0.169	0.12	0.309	0.324
DSP	70	128	40	80	112
LUT	1722	4173	1201	2747	14670

TABLE 3.3 – Résultats des implémentations de l'architecture semi-parallèle sur la famille de FPGA Intel-Altera

Dans les tableaux 3.2 et 3.3 respectivement, nous donnons un rapport des résultats de notre architecture semi-parallèle pour Xilinx et Intel-Altera FPGA après placement et routage. Les résultats des mises en œuvre de nos architectures séquentielles sont présentés dans les figures 3.4 et 3.5.

Comme attendu, l'architecture semi-parallèle est plus rapide que l'architecture séquentielle tandis que l'architecture séquentielle permet d'obtenir la plus petite surface. Nous observons que le FPGA Xilinx fournit les calculs les plus rapides pour la multiplication modulaire dans la représentation AMNS par rapport à la famille Altera.

Famille	Xilinx				
FPGA	Zynq		Virtex 6		
p bit	128	256	128	256	512
w	36		36		72
Fq(Mhz)	283.2	255.7	309.5	189.7	224.1
Cycles	23	47	23	47	47
Temps μs	0.081	0.183	0.074	0.247	0.209
DSP	51	53	51	91	216
LUT	791	2827	791	1740	24763

TABLE 3.4 – Résultats des implémentation matérielle de l'architecture séquentielle dans la famille FPGA de Xilinx

Famille	Altera				
FPGA	Aria II		Cyclone V		
p bit	128	256	128	256	512
w	36		36		72
Fq(Mhz)	237.3	213.3	157.2	123.4	82.7
Cycles	23	47	23	47	47
Temps μs	0.096	0.22	0.146	0.38	0.568
DSP	42	80	28	52	112
LUT	1267	2423	679	1304	14283

TABLE 3.5 – Résultats des implémentation matérielle de l'architecture séquentielle dans la famille FPGA de Intel-Altera

3.3.1 Analyse des résultats

À notre connaissance, notre implémentation est la première implémentation VHDL d'une architecture FPGA d'une multiplication modulaire utilisant les AMNS comme système de représentation. Nous avons comparé notre conception à certains travaux publiés précédemment :

— La première conception à laquelle nous comparons est détaillée dans [OBPV03].

Il s'agit d'une architecture systolique pour la multiplication modulaire de Montgomery.

— La deuxième conception est également une architecture systolique qui est basée sur la multiplication modulaire de Montgomery du CIOS par [MML+16]. Elle utilise également des nombres binaires.

— La troisième conception est détaillée dans [BT15]. Son originalité est qu'il utilise les *Residue Number Systems* (RNS) comme système de représentation de nombres entiers. Cette architecture est basée sur des composants cox-rower introduits dans [KKSS00].

— La conception publiée dans [RK14] utilise la représentation "carry-save" dans les résultats intermédiaires.

— L'architecture systolique décrite dans [RK16] utilise la représentation binaire des chiffres signés afin de minimiser le nombre de produits partiels non nuls dans la multiplication modulaire.

— De même, l'architecture présentée dans [RK15] utilise une représentation en chiffres binaires signés.

La comparaison de la conception des FPGA est une tâche difficile car les technologies utilisées sont hétérogènes. Dans cette comparaison, nous avons détaillé le nombre de registres, le nombre de blocs DSP, LUT et le nombre de cycles nécessaires pour effectuer la multiplication modulaire et le délai total de l'implantation. Certaines architectures utilisent des blocs DSP, tandis que d'autres n'utilisent que des blocs LUT. À des fins de comparaison, nous avons unifié la mesure de la surface en calculant une valeur estimée de $LUT_{eq} = nLUT + (nDSP \times r)$, où $nLUT$ est le nombre de ressources LUT, $nDSP$ est le nombre de blocs DSP et r est le rapport entre le nombre de LUT et le nombre de blocs DSP disponibles sur le dispositif cible. Ainsi, le produit surface-délai (ADP) est calculé comme suit : $LUT_{eq} \times Time(\mu s)$.

Architecture	FPGA	p size	DSP	LUT	Cycle	Time (μs)	ADP
Semi-parallel	A7	128	60	1563	19	0.077	901.1
		256	120	2728	33	0.165	3796.3
		512	188	29985	33	0.204	12598.4
	V6	128	60	1035	19	0.079	844.9
		256	120	2738	33	0.146	3220.4
		512	236	18964	33	0.172	9797.1

Séquentielle	A7	128	51	790	23	0.12	1129
		256	91	1718	47	0.242	4137.4
		512	176	37138	47	0.258	17255.5
	V6	128	51	791	23	0.074	666.1
		256	91	1740	47	0.24	3933.8
		512	216	24763	47	0.209	12443.6
Mrabet et al. (2016)	A7	128	19	355	33	0.166	591.9
		256	33	809	33	0.311	1986
		512	87	2650	33	0.507	8797.9
Bigou and Tisserand (2015)	K7	192	18	999	58	0.213	864.5
		384	41	2111	58	0.324	2942.2
		512	56	8757	66	0.374	6835.5
	V5	192	15	1447	58	0.295	741
		384	42	2256	58	0.467	2446.1
		512	57	10877	66	0.536	7999.2
Ors et al. (2015)	V-E	128	-	806	388	1.807	1456.4
		256	-	1548	772	7.686	11897.9
		512	-	2972	1450	16.17	48057.2
Rezai and Keshavarzi (2014)	V5	512	-	3048	-	0.449	1400
Rezai and Keshavarzi (2015)	V5	512	-	6091	-	0.851	5200

TABLE 3.6: Comparaison de nos implémentations avec l'état de l'art pour un module de p de 128 bits jusqu'au 512 bits.

Dans le tableau 3.6 nous désignons par A et V respectivement le FPGA Artix et Virtex.

Architecture	FPGA	LUTeq	Temps (μs)	ADP	Débit
Semi-parallèle	A7	23008	0.165	3796.32	1544.3
	V6	22058	0.146	3220.468	1751.8
Séquentielle	A7	17097	0.242	4137.47	1055.5
	V6	16391	0.24	3933.84	1033.5
Mrabet et al. (2016)	A7	6386	0.311	1986.04	822.3

Bigou and Tisserand (2015)	K7	9081	0.324	2942.24	1185.1
	V5	5238	0.467	2446.146	820.9
Rezai and Keshavarzi (2015)	V-E	1548	7.686	11897.928	33.3

TABLE 3.7: Comparaison de l'ADP et du débit pour le module de p égale à 256 bits.

Nous présentons dans le tableau 3.6 les résultats de l'implémentation de nos architectures pour les modules 128 bits, 256 bits et 512 bits.

Nous avons présenté aussi les résultats d'implantation sur FPGA des travaux que nous avons cité dans la partie au-dessus. Nous avons choisi cette plage de taille de module, afin de cibler les implantations sur les ECC ([HM11]).

La comparaison directe des résultats d'implantation rapportés dans le tableau 3.6 est une étape un peu difficile en raison de la diversité des valeurs de p. Pour cela, nous mettons en évidence les résultats en matière de surface, de temps et de débit pour une taille du module p égal à 256 bits et à 1024 bits. Le tableau 3.7 représente les résultats pour les modules p de 256 bits. Le tableau 3.8 présente les résultats pour les grandes tailles cryptographiques qui ciblent l'algorithme RSA ([RSA78a]). Dans ce tableau, les modules cibles sont e 1024 bits.

Dans cette comparaison, nousd avons détaillé le ADP (Area-Delay Product), le débit (Throughput) et le délais d'implantation de la multiplication modulaire pour un module égal à 1024 bits. Notre implémentation semi-parallèle permet d'obtenir un débit plus élevé que la version séquentielle. Bien que l'ADP du semi-parallèle soit meilleur que celui de l'architecture séquentielle, il ne s'approche pas du minimum de l'ADP dans toutes les technologies. Les performances de l'architecture semi-parallèle sont généralement meilleures que celles de l'architecture séquentielle.

Il semble que nos modèles soient plus rapides que les autres pour les modules 256 bits ainsi que pour les modules 1024. En conséquence, le débit est le plus élevé pour toutes les tailles de modules. Ceci est principalement dû au nombre de blocs DSP qui augmentent considérablement la vitesse de l'architecture, au prix de la surface.Par rapport à la conception décrite dans Ors et al.(2015) [OBPV03], ce délicat de comparés avec Ors et al. parce que ce n'est pas la même technologie. Toutefois, nous observons qu'avec une technologie FPGA plus avancée, nous arrivons à être sensiblement plus rapide.

Par rapport à l'architecture de Mrabet et al.(2016) [MML+16] qui est une

architecture matérielle systolique de multiplication modulaire de Montgomery, notre conception est plus rapide de 2.15 fois avec un meilleur débit et plus élevée de 1.8 fois que les résultats présenté dans Mrabet et al.(2016) [MML+16].

Architecture	FPGA	Surface (LUT_{eq})	Temps (μs)	ADP	Débit (Mb/s)
Semi-parallèle	V7	124151	0.43	53384	2381.39
Séquentielle	V7	55667	0.598	33288	1712.37
Rezai and Keshavarzi (2014)	V5	6105	0.883	5390	1159.6
Rezai and Keshavarzi (2016)	V6	1125	2.56	2880	356
Rezai and Keshavarzi (2015)	V5	6091	0.851	5180	1203.3

TABLE 3.8 – Comparaison de résultat l'ADP et du débit pour un module p – 1024-bits.

La conception de Bigou andTisserand (2015) [BT15] est une conception originale qui utilise un système modulaire de représentation. Par rapport à cette architecture, nos conceptions restent plus rapides de 1.96 fois mais plus grandes de 2.53 fois. Le débit de notre architecture est plus élevé de 1.30 que les résultats présenté dans Bigou andTisserand (2015) [BT15].

Si l'on considère la comparaison avec les architectures de multiplication modulaire décrites dans Rezai and Keshavarzi(2016) [RK16], Rezai and Keshavarzi(2014) [RK14] et Rezai and Keshavarzi(2015) [RK15], notre conception séquentielle est respectivement de 5.95, 2.05 et 1.97 fois plus rapide au prix d'un plus grand nombre de ressources. Nous pouvons constater que le débit obtenu par ces mises en œuvre est nettement inférieur à celui que nous sommes en mesure d'obtenir avec les mises en œuvre semi-parallèles et séquentielles que nous proposons.

Quelle que soit la taille du module p, notre conception séquentielle et semi-parallèle à une caractéristique intéressante : le nombre de boucles est indépendant de la taille du module. Cela est dû à la structure que nous avons proposée.

3.4 Conclusion et perspectives

Dans ce chapitre, nous avons proposé une architecture matérielle rapide qui réalise une multiplication modulaire sur de grands nombre sur le corps fini de caractéristique premier $\mathbb{F}_p$, qui utilise des représentations AMNS afin d'accélérer

la multiplication modulaire. Pour mener bien à l'implantation de cette architecture, nous avons proposé deux architectures matérielles : une architecture semi-parallèle et une architecture séquentielle. Notre architecture semi-parallèle offre les mises en œuvre les plus rapides.

Comme cette implantation est la première matérielle de la multiplication modulaire dans AMNS, nous comparons en deuxième lieu nos résultats avec les multiplications modulaires existantes comme le CIOS systolique, les architectures systoliques utilisant des représentations à sauvegarde de retenue ou à chiffres signés pour les résultats intermédiaires et l'architecture conçue pour la multiplication modulaire RNS. La comparaison de nos résultats avec l'état de l'art met en évidence que nous proposons l'implantation la plus rapide de la multiplication modulaire pour les applications à grand nombre avec le débit le plus élevé. Ces résultats nous encouragent afin de concevoir d'autres architectures qui donnent des résultats améliorés.

4 Conception et implantation matérielle de la randomisation dans AMNS

Sommaire

Ces dernières années, les systèmes cryptographiques sont devenus la cible de diverses attaques par canaux auxiliaires. Par exemple, dans [KJJ99], les auteurs ont montré que l'implantation matérielle d'un système cryptographique peut être vulnérable pour les attaques par canaux auxiliaires. Dans cet article, Kocher a souligné que l'exécution de l'algorithme de chiffrement peut révéler des informations sur la clé utilisée. Le principe de ces attaques est basé sur l'observation de la consommation énergétique pour en extraire des informations. Il s'agit ici de l'analyse de puissance simple (SPA).

Dans ce chapitre, nous nous concentrons sur la randomisation de la multiplication modulaire de Montgomery en AMNS, avec pour objectif d'avoir un premier résultat sur le coût d'implantation de l'algorithme de randomisation. A notre connaissance, cela n'a été jamais implémenté sur FPGA. Il serait intéressant de comparer les résultats d'implantation avec d'autres techniques de

randomisation. Plus précisément, nous allons concevoir une architecture qui peut être utilisée pour protéger cette multiplication scalaire contre les attaques par canaux auxiliaires.

Il existe deux méthodes complémentaires pour la randomisation des données. La première est la randomisation de la multiplication modulaire en AMNS. La deuxième est la randomisation des coordonnées initiales du point P utilisé pour la multiplication scalaire sur courbe elliptique.

Lors du choix d'un algorithme de randomisation de la multiplication modulaire pour une implantation sécurisée, nous devons tenir compte de l'impact sur les performances de l'implantation matérielle du système cryptographique. En effet, l'implantation de l'algorithme de la multiplication modulaire randomisé entraîne une augmentation des ressources utilisées en FPGA. Pour compenser cette augmentation de la surface, nous avons proposé une architecture qui utilise les multiplicateurs dédiés et les blocs de RAM sur le FPGA. Cette architecture est basée sur la randomisation de l'algorithme de multiplication modulaire sur le système AMNS [DDV20].

Ce chapitre est organisé comme suit. Nous commençons par présenter un état de l'art de la randomisation de l'arithmétique modulaire. Ensuite, nous nous intéressons à la randomisation de la multiplication modulaire en AMNS. Enfin, nous présentons l'architecture matérielle que nous avons conçue pour la randomisation de la multiplication modulaire en AMNS sur FPGA. Ainsi que le coût de sa première implantation sur FPGA.

4.1 Multiplication modulaire randomisée : principe et état de l'art

A partir des années 1990, des contre-mesures ont été prises en compte afin d'empêcher les pertes d'information dues à la consommation d'énergie.

L'algorithme de randomisation de la multiplication modulaire en AMNS a été détaillé dans la section II.2 du chapitre 2 dans la thèse de Jéremy Marrez [jM19]. En outre, il a été étudié dans la section 3.3.3 du chapitre 3 de la thèse de Yssouf Dosso [Dos20]. Le principe de la randomisation est simple, il suffit d'ajouter une donnée aléatoire à l'entrée de la multiplication modulaire. Cette donnée aléatoire admet certaines propriétés. En effet, nous nous appuyons sur le fait que le système est redondant, c'est-à-dire, un entier possède avoir plusieurs représentants dans $\mathbb{Z}/p\mathbb{Z}$. En particulier, l'élément neutre peut avoir plusieurs redondants et il suffit d'en choisir un représentant aléatoire de l'élément neutre

pour construire un ensemble de représentants AMNS d'une valeur p. De ce fait, cette méthode permet de masquer les données confidentielles en ajoutant un représentant de l'élément neutre choisi aléatoirement. Ce là aboutit à la génération d'une représentation aléatoire de la donnée. Plus précisément, la randomisation se réalise en deux phases, la première consiste à générer un AMNS dans $\mathbb{Z}/p\mathbb{Z}$ tel que chaque élément admet au minimum $(2z+1)^n$ représentations distinctes, avec $z \in \mathbb{N}$ qui est un paramètre de l'ensemble $H = \{Z \in \mathbb{Z}/p\mathbb{Z}$, tel que $||Z||_\infty \leq z\}$.

La deuxième phase réside sur l'opération de la randomisation, qui repose sur la génération aléatoire des représentants en AMNS qui se fait à l'aide du polynôme de randomisation $Z \in H$.

Dans cette partie, nous présentons deux algorithmes de randomisations qui sont utilisés dans l'état de l'art, afin de randomiser la multiplication modulaire sur l'AMNS. Ces deux algorithmes ont été présentés par Didier et al. dans [DDEM+19b]. Le détail du mécanisme de randomisation se passe dans la partie réduction interne ce qui explique que le premier algorithme est basé sur la méthode de réduction interne du type Montgomery présenté dans le chapitre 1. Le deuxième algorithme est basé sur la méthode de Babai détaillée dans la thèse de Jéremy Marrez [jM19]. L'algorithme de Montgomery semble être plus efficace et plus rapide que celui de Babai. Nous nous intéressons donc uniquement à l'algorithme de Montgomery.

4.1.1 Algorithme de Montgomery-like

L'algorithme 13 est décrit dans [DDEM+19b]. Cet algorithme randomise la multiplication modulaire $A \times B$ en AMNS, avec A et B deux éléments de l'AMNS. Nous notons par $\mathcal{B} = (p, n, \gamma, \rho)$ un système AMNS.

L'idée générale est de randomiser l'opérande B afin que toute la multiplication modulaire soit randomisée. Dans l'algorithme 13, cette randomisation est faite aux lignes 1, 2 , 3.

Le polynôme randomisé B' est un représentant de B dans $\mathcal{B}$. Il est calculé à partir du polynôme de randomisation Z, tel que
$Z(X) = z_0 + z_1X + \cdots + z_{n-1}X^{n-1}$, avec $-z \leq z_i \leq z$.

La génération du polynôme de randomisation Z est faite avec la fonction **RandomPoly**. Cette fonction permet de générer aléatoirement un polynôme à partir d'un entier $z \in \mathbb{N}$ tel que $\|Z\|_\infty \leq z$ et $deg\ Z < n$. Dès que nous fixons z, nous aurons exactement $(2z+1)^n$ représentant distinct pour chaque polynôme de $\mathbb{Z}/p\mathbb{Z}$.

Les lignes 4 et 5 de l'algorithme représentent la réduction interne et la réduction externe de la multiplication modulaire classique dans AMNS.

L'algorithme 13 permet de générer $(2z+1)^n$ représentant distinctes pour tous les polynômes possible à générer par la fonction **RandomPoly**. Pour que l'algorithme de randomisation soit efficace, il est essentiel que les bornes ρ et ϕ soient tels que :

$$\rho \geq w\ ||M||_\infty\ (2+2z) \text{ et } \phi \geq \frac{3\rho^2}{2||M||_\infty}(\delta+1)^2$$

avec $w = 1+(n-1)|\lambda|$, $E(X) = X^n - \lambda$ et δ est le nombre maximum d'additions successives d'éléments de $\mathcal{B}$. Cette valeur change selon l'application dédiée, prenons l'exemple des formules d'addition et de doublement de points sur les courbes elliptique. La valeur de δ est égale au nombre des additions et de doublement de points qui sont connus.

Algorithm 13 Multiplication modulaire randomisée dans le système AMNS via Montgomery-like

Entrées: $\mathcal{B} = (p,\ n,\ \gamma,\ \rho, E)$ and
$A, B \in \mathcal{B}$
Sorties: $R(\gamma) = A(\gamma)B(\gamma)\phi^{-1} \pmod p$
Data : $\phi = 2^j,\ j \geq 1\,;\ z \in \mathbb{N}$
$M \in \mathcal{B}$, tels que :
$M(\gamma) \equiv 0 \pmod p$ et
$\gcd(\phi,\ \text{resultant}(E,\ M)) = 1\,;$
$M' = -M^{-1} \bmod (E, \phi)\,;$

1: $Z \leftarrow$ **RandomPoly**(z)
2: $J \leftarrow Z \times M \mod E$
3: $B' \leftarrow B + J$
4: $C \leftarrow (A \times B') \mod E$
5: $S \leftarrow$ **RedCoeff**(C) **#en utilisant la méthode de Montgomery-like**
6: $R \leftarrow S + 2 \times J$
7: **return** R

4.2 Modélisation et Implantation en description RTL

Après avoir choisi la méthode et l'algorithme de randomisation dans la partie précédente. Dans cette partie, nous proposerons la conception matérielle de l'algorithme de randomisation. Afin d'avoir une implémentation efficace et op-

timisée. Précisément, nous intégrons l'IP matérielle conçue dans le chapitre 3. La première est une architecture séquentielle et la seconde est une architecture semi-pipeline. Notre étude comparative au chapitre 3 montre que l'architecture semi-pipeline est la méthode la plus rapide.

C'est pour cela, nous avons intégré l'IP matérielle semi-pipeline dans la conception de l'architecture de multiplication modulaire randomisée.

4.2.1 Modélisation du système

Notre implémentation matérielle de la multiplication modulaire randomisé dans le système de représentation AMNS est basée sur l'algorithme 13. Cet algorithme peut être développé comme suit :

Algorithm 14 Multiplication modulaire randomisée - étapes principales

Entrées: $A \in \mathcal{B}$, $B \in \mathcal{B}$ and $\mathcal{B} = (p,\ n,\ \gamma,\ \rho, E)$

Sorties: $S \in \mathcal{B}$

$$J \leftarrow Z \times M \tag{4.1}$$
$$B' \leftarrow B + J \tag{4.2}$$
$$C \leftarrow A \times B' \tag{4.3}$$
$$C' \leftarrow C \bmod E \tag{4.4}$$
$$Q \leftarrow C' \times M' \bmod (E, 2^r) \tag{4.5}$$
$$R \leftarrow ((Q \times M)) \bmod E \tag{4.6}$$
$$S' \leftarrow (C' + R)/2^r \tag{4.7}$$
$$S \leftarrow 2J + S' \tag{4.8}$$

Nous rappelons que les entréesA, B, M, M' et Z sont des polynômes de degré n-1. Ainsi, le polynôme A a des coefficients n noté a_i pour $i \in \{0, \cdots, n\}$.

Le polynôme M est le polynôme de réduction interne et l'inverse modulaire de son modulo opposé $(E, 2^r)$ est M'. Notons que pour des raisons d'efficacité, nous choisissons $\phi = 2^r$. Le polynôme de réduction externe E est un polynôme creux de la forme $E(X) = X^n - \lambda$.

Le polynôme Z est un polynôme généré aléatoirement à coefficients dans l'ensemble $\{-z, ..., z\}$. L'architecture que nous avons conçue prend le polynôme Z en entrée. Par la suite, nous ne nous intéresserons pas à cette partie sur la génération du polynôme de randomisation Z.

Les polynômes C, C', R et S' sont des polynômes intermédiaires. L'algorithme 14 génère le polynôme de sortie S qui est le résultat de la multiplication modulaire randomisé.

Nous utilisons la mémoire BRAM de le FPGA afin de sauvegarder les coefficients des polynômes A, B, C, R, S' et les coefficients des polynômes constants M, M' et Z. Ainsi que le résultat final est réécrit dans la même mémoire.

Nous proposons une conception de notre architecture dans la Fig. 4.1. Dans cette figure, l'opérateur *Multiplication modulaire randomisée*, traite les étapes de (1) jusqu'a (8) de l'algorithme de multiplication AMNS.

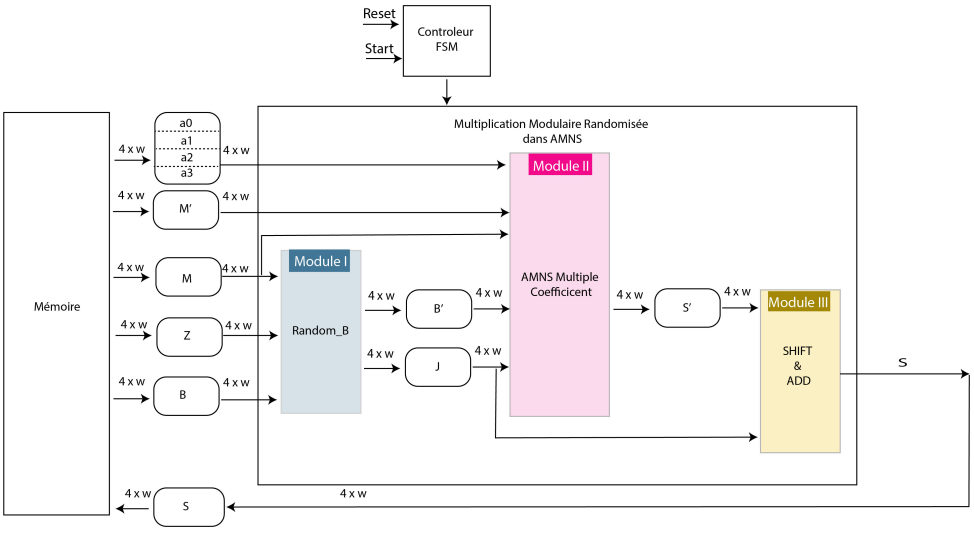

FIGURE 4.1 – Architecture de la multiplication Modulaire par Randomisation (MMR)

Ce bloc se compose de trois modules. Le premier module est nommé *Random_B*, cette fonction randomise le polynôme B. Par la suite, le module prend les coefficients des polynômes de randomisation Z, B et M en entrée. Ce module traite des étapes (4.1) et (4.2). Il lit le coefficient de chaque valeur d'entrée b_j, m_j et z_j dans la mémoire, traite les résultats intermédiaires J, B' et les stocke dans des registres locaux dédiés.

Tandis que le deuxième module qui est nommé **AMNS Multiple coefficient** traite les étapes (4.3) au (4.6) de l'algorithme 13. Il prend en entrée les coefficients des polynômes B', A, M et M'. Ce module génère deux sorties R et C' qui seront traités dans le troisième module. En fait, ce module est le même que celui que nous avons décrit dans le chapitre 3, section 4.2.2.1. en effet, c'est l'architecture semi-parallèle que nous intégrons dans l'architecture de la randomisation.

Finalement, le troisième module est nommé **SHIFT&ADD** contient deux opérations ; $SHIFT$ et ADD. Ces opérations sont utilisées afin d'exécuter les étapes (4.7) et (4.8) de l'algorithme 15. Par la suite, nous obtenons la sortie S qui est le résultat de la multiplication modulaire randomisée.

Nous notons que l'exemple que nous présentons dans la figure 4.1 est pour la valeur du module p de 128 bits. Les valeurs que nous avons fournies dans l'exemple ont été choisies pour illustrer un exemple simple. Pour cela, nous donnons quatre coefficients de taille w dans chaque polynôme. Cependant, l'architecture que nous avons implémentée a un module $p = 256$ bits.

4.2.2 Implémentation Matérielle

Dans cette partie, nous présentons une architecture qui peut être utilisée pour calculer une multiplication modulaire randomisée dans le système de représentation AMNS sur FPGA. L'architecture que nous avons conçue est semi-parallèle, dans laquelle nous intégrons l'architecture semi-parallèle de multiplication modulaire qui a été introduite au chapitre 3. Dans la partie 4.2.2.1, nous présentons notre architecture semi-parallèle pour l'implantation de la randomisation.

4.2.2.1 Architecture Semi-parallèle de la randomisation

Dans cette architecture, les étapes décrites dans la partie 4.2.1 sont programmées comme indiqué dans la figure 4.2. En effet, cette figure est un exemple d'ordonnancement de multiplication modulaire randomisée dans le système de représentation AMNS, ses entrées A et B ont quatre coefficients de 36 bits, et le module p est écrit sur 128 bits.

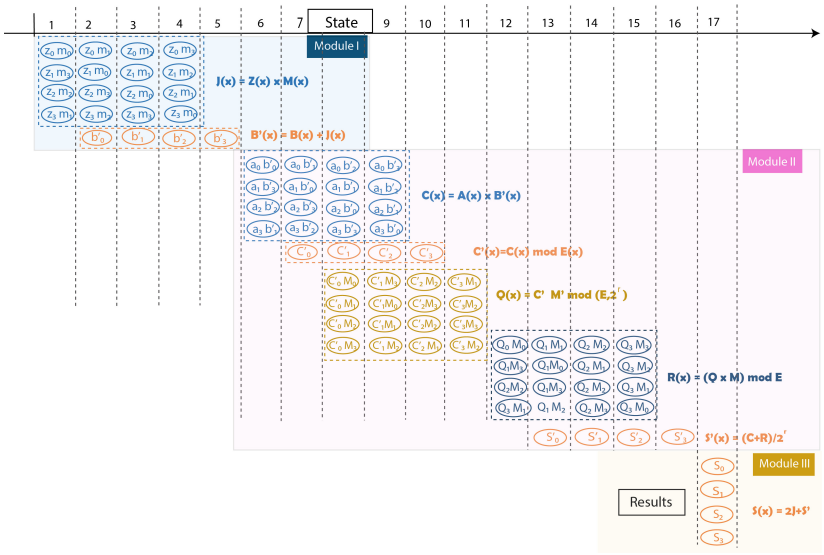

FIGURE 4.2 – Ordonnancement de l'architecture pipeline de la multiplication modulaire randomisée

L'exécution de notre architecture se fait sur 17 états. L'ordonnancement de la multiplication modulaire randomisée est repartie sur trois parties indispensables ; la première partie est le module de randomisation **Random_B** et la deuxième partie est celle de la réduction modulaire **AMNS Multiple Coefficient**. Finalement le dernier fait l'addition et le décalage **SHIFT&ADD**.

Le module I prends 5 états, afin de s'exécute les équation (4.1) et (4.2) de l'algorithme 15. Dans son premier état, nous exécutons quatre multiplications en parallèle comme le montre la figure 4.2. Les variables z_0, z_1, z_2, z_3 et m_0, m_1, m_2, m_3, m_4 représentent respectivement les coefficients du polynôme Z et M. Par la suite, l'exécution de l'étape (4.1) de l'algorithme 15 ($J \leftarrow Z \times M$) est réalisé sur quatre états. L'exécution de l'étape (4.2) de l'algorithme 15 ($B' \leftarrow B + J$) commence à partir du deuxième cycle. En outre, les coefficients du polynôme B' seront généré consécutivement sur quatre cycles. Ce qui nous permet d'exécuter l'étape (4.2) en parallèle avec l'étape (4.1).

Le module II, s'exécute en 11 états comme illustre la figure 4.2. Ce module exécute les étapes de la réduction modulaire de la multiplication modulaire. Nous renvoyons le lecteur vers la partie , afin d'avoir plus des détailles sur l'exécution de ce module.

Le module III, exécute l'étape (4.8) ($S \leftarrow 2J + S'$) dans un seul cycle.

Les architectures matérielles des ces trois modules sont détailles dans la partie ci-dessous.

A) **Architecture matérielle du module I : Random_B**

L'architecture matérielle de ce module est illustré dans la figure 4.3.

Ce module est chargé des étapes (4.1) et (4.2) de l'algorithme 14.

En premier lieux, le calcul de l'étape (4.1) se fait en deux étapes comme suit :

D'abord, il charge les coefficients du polynôme Z ainsi que les coefficients du polynôme M. Ces deux polynômes seront fourni en tant que des données en entrée. Ensuite, ce module utilisera les quatre multiplieurs **MULTIPLIER** (36 bits $\times$ 40 bits), afin de calculer le produit des coefficients $m_i \times Z_j$ (40 bits $\times$ 36 bits). Chaque multiplieur générera un bout de coefficient J_i du polynôme J. finalement, nous utiliserons un additionneur $ADDER$, pour calculer la somme des bouts des coefficients J_{ij} en vue de résulter les coefficients J_i d'une taille égale à 76 bits. Les résultats de l'addition des coefficients génèrent consécutivement les quarte coefficients

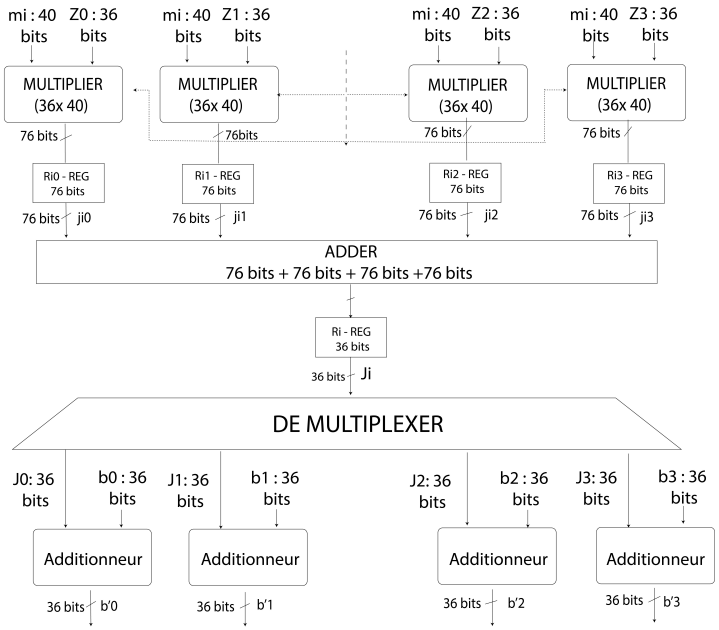

FIGURE 4.3 – Architecture du module I : randomisation de B, calcul des étapes (4.1) et (4.2)

J_i du polynôme J.

En deuxième lieu, le calcul de l'étape (4.2) se fait aussi en deux étapes. La première étape charge les coefficients J_i consécutivement en entrées de l'*Additionneur* à travers un $DEMULTIPLEXER$. La deuxième étape exécute consécutivement l'opération d'addition des coefficients $(b_i + J_j)$, en vue de calculer les coefficients b'_i du polynôme B.

Le résultat est enregistré dans les registres J_{ij} qui seront pris en entré dans les blocs d'addition qui chargent les coefficients de J_i dans des registres de taille 76 bits. La sortie est connectée à un dé-multiplexeur qui produit tous les coefficients de J. La dernière étape dans ce module sert à calculer l'addition $J + B$. Cette étape est composée de quatre blocs d'additions de 36 bits, qui ont en entrée des coefficients de 36 bits des polynômes B et J.

B) **Architecture du module II :AMNS Multiple coefficient**

Ce module exécute les étapes (4.3), (4.4), (4.5), (4.6), (4.7) et (4.8) de l'algorithme 14, afin de calculer la multiplication modulaire en AMNS de $A \times B'$. Ainsi, les coefficients du polynôme B' seront fournis comme entrée dans le bloc de multiplication modulaire de $A \times B'$. Ce bloc représente le module II. En effet, la conception de cette multiplication modulaire est la

même que nous avons détaillée dans la section 4.2.2.1.

C) **Architecture matérielle du module III :**

L'architecture matérielle du module III calcul l'étape (4.8) de l'algorithme 14. Cette étape est décrite dans la figure 4.4. En fait, le module se compose de quatre registres à décalage gauche et de quatre additionneurs. Les quatre registres à décalage ont en entrées les coefficients de 36 bits

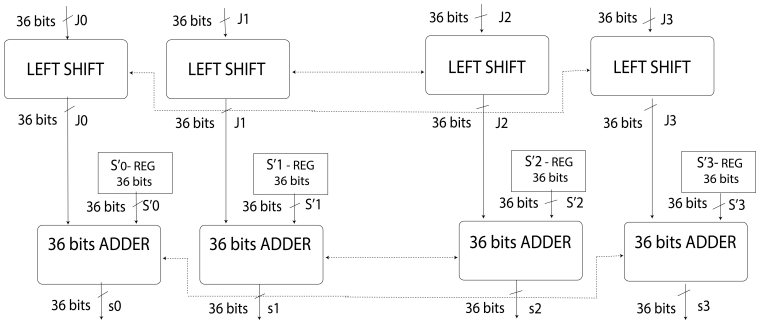

FIGURE 4.4 – Architecture du module III : calcul de l'étape (4.8)

du polynôme J. Dans cette étape, une multiplication par 2 est effectuée qui est représenté par les blocs $LEFT\ SHIFT$. Ensuite, nous utilisons quatre blocs $ADDER$ qui prennent les coefficients du polynôme S' de 36 bits et les coefficients du $2J$ de taille 36 bits en entrée.

D) **Main Contrôleur MC** Cette conception est contrôlée par une machine à état finis (FSM). La figure 4.5 illustre la FSM de notre architecture pour $w = 36$ bits et un module p de 256 bits. Elle est composée de 24 états, l'état 1 étant l'état de départ.

Le **module I** qui sert à randomiser le polynôme B commence par l'état 1 et se termine par l'état 10. Ces états permettent de calculer les coefficients du polynôme B' et de les charger dans le registre BRAM. Cela correspond aux étapes (4.1) et (4.2) de l'algorithme 14. À l'état 11, il faut vérifier si le processus du module de randomisation de B est terminé. Si c'est "Oui", alors FSM passe au module "Multiplication modulaire", si c'est "Non", alors FSM revient au point de départ.

Le passage vers le **module II** de la "Multiplication modulaire" est exécuté dans un sous-programme, qui commence à l'état 11 et se termine à l'état 21. Ces états permettent de calculer les coefficients du polynôme B' et

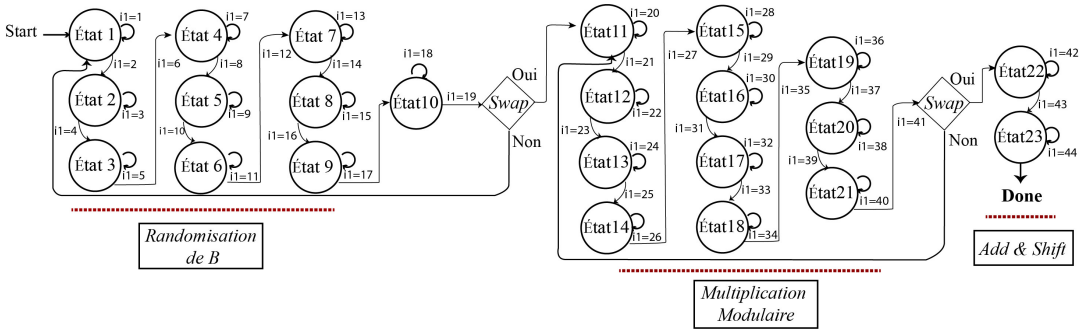

FIGURE 4.5 – Machine à état finis pour l'architecture semi-parallèle de la randomisation avec $w = 36$ bits et a $p = 128$ bits

de les charger dans le registre BRAM. Cela correspond aux étapes (4.3) au (4.7) de l'algorithme 14. Un signal *done_mult* = 1 est émis pour indiquer que le processus de multiplication modulaire est terminé et que les résultats sont prêt à être utilisé par le sous-programme suivant.

À l'état 22, la condition est faite pour vérifier la valeur du bit done_mult. Si c'est égale à 1, alors la FSM passe au module "ADD & SHIFT". Si c'est 0, alors la FSM revient à l'état 11.

Le **module III** "ADD & SHIFT" est exécuté dans un sous-programme, qui commence à l'état 22 et se termine à l'état 23.

Dans l'état 23, la FSM transmis un signal *done* pour indiquer que le processus de multiplication modulaire randomisée est terminé et que les coordonnées du point S sont obtenues.

Remarque 4.1. Il faut noter que chaque état de la machine à état fini prend deux cycles pour son exécution. Ce qui fait que notre architecture de la multiplication modulaire randomisée se déroule sur 48 cycles.

4.3 Résultats expérimentaux

Nous avons complètement implémenté et validé l'architecture semi-parallèle pour la multiplication modulaire randomisée dans AMNS sur FPGA. Les résultats sont donnés pour 2 familles de FPGA : Virtex 7 (xc7vx690tffg 1761 -2) et Cyclone V (5cgXFC7DF 31 C8ES). Nous utilisons les outils ISE 2020.1 pour le FPGA Xilinx et Quartus II 12.1 pour le FPGA Altera. Nous avons choisi deux familles différentes de FPGA, afin de confirmer nos résultats avec deux outils différents. Afin de pouvoir présenter une comparaison équitable avec l'état de l'art de l'implantation. Nous mettons en œuvre les architectures que nous

Famille	FPGA	LUT	DSP	Fréq.	speed (μs)
Xilinx	V7	8960	282	223.81	0.209
Altera	C5	10518	144	108.8	0.422

TABLE 4.1 – Résultats d'implantation materielle de la multiplication modulaire randomiser sur AMNS pour p=256 et w=64 en utilisant 47 cycle.

proposons sur un FPGA Virtex-7 en utilisant le VHDL. Les résultats de notre implémentation sont générés en utilisant Xilinx Vivado 2020.1. Nous implémentons également nos architectures sur un FPGA Cyclone 5 en utilisant VHDL avec Quartus 12.1 et l'option de synthèse par défaut.

4.3.1 Analyse des résultats

À notre connaissance, notre implantation est la première en VHDL de la multiplication modulaire randomisée en utilisant AMNS comme système de représentation. Nous avons comparé le résultat de l'implantation avec l'implantation d'une multiplication modulaire en AMNS sans randomisation pour une taille de p de 256 bits.

Nous présentons dans le tableau 4.2 les résultats de l'implantation de nos archi-

Architecture	Famille	FPGA	Surface (LUT_{eq})	temps (μs)	ADP	Débit (Mb/s)
MM Randomisation	Xilinx	V7	17443.5	0.209	3645.69	1224.88
	Altera	C5	62593.38	0.422	26414.4	606.63
MM sans Randomisation	Xilinx	V7	10979.66	0.146	1603.03	1753.12
	Altera	C5	31677.76	0.309	9788.42	828.47

TABLE 4.2 – Comparaison des résultats d'implantation de la multiplication modulaire randomisée et de la multiplication modulaire sans randomisation dans l'AMNS, p=256 bits

tectures par rapport à l'état de l'art pour une taille de module égale à 256 bits. Une telle plage de taille de module vise l'implantation sur l'ECC ([HM11]). Nous mettons en évidence les résultats en termes de surface, de temps et de débit pour un module p de 256 bits.

Notre implantation de la multiplication modulaire randomisée (MMR) sur

la virtexe 7 dans le tableau 4.2, permet d'obtenir un temps d'exécution très proche de celui de l'implantation d'une multiplication modulaire dans le système de représentation AMNS sans randomisation. En effet, notre architecture randomisée est 1.4 fois plus lente que l'architecture sans randomisation.

Nous observons le même impact sur le FPGA Altera C5. En effet, l'implantation de la MMR est plus lente de 1.33 fois par rapport à la multiplication modulaire sans randomisation.

La comparaison du nombre de slices LUTs sur la virtex 7 de la MM avec randomisation et MM sans randomisation, montre que la randomisation utilise 13% de plus des LUTS que l'architecture sans randomisation. Ce qui introduit une augmentation de la valeur de l'ADP de 43.97%. Ce qui explique la diminution de la valeur du débit de 14.31%.

Le temps d'exécution de l'implantation de l'architecture de la multiplication modulaire randomisé sur la famille Cyclone C5 est plus lent de 20.01% que celle sans randomisation. Ce résultat est justifié puisque la famille Virtex 7 est plus performante que le cyclone 5 au niveau du nombre de ressources et de la fréquence.

nous remarquons aussi que le nombre des LUTs de l'implantation de la multiplication modulaire randomisé sur Cyclone 5 est plus grande de 50% que celle sans randomisation. Les facteurs du temps et de surface influent les résultats trouvé pour l'ADP et le débit de la même façon que l'implantation sur Virtex 7.

4.4 Conclusion et perspectives

Dans ce chapitre, nous présentons une architecture matérielle rapide qui réalise une multiplication modulaire randomisée (MMR) sur le corps finis de caractéristiques premiers $\mathbb{F}_p$, qui utilise des représentations en AMNS afin de sécuriser le crypto-système dans l'ECC. L'architecture que nous avons conçue pour la randomisation intègre l'IP matérielle semi-parallèle de la multiplication modulaire que nous avons présenté au chapitre 3.

Comme nous proposons la première implantation matérielle de la multiplication modulaire randomisé dans le système de représentation AMNS, nous avons comparé nos résultats avec la multiplication que nous avons introduite au chapitre 3. L'analyse de nos résultats met en évidence que nous proposons une implantation rapide de la MMR pour les applications dans le corps finis.

Ces premiers résultats nous encouragent à explorer d'autres pistes pour améliorer nos résultats. Ainsi que de tester la sûreté de nos implantations envers les attaques par canaux cachés. Au niveau des ECC, il pourrait être intéressant d'assurer la résistance de la multiplication scalaire en ECC contre les attaques SCAs, en utilisant la randomisation de la multiplication modulaire en AMNS.

5 Multiplication scalaire sur les courbes elliptiques

Sommaire

Un des axes de cette thèse est de concevoir un crypto-système utilisant les courbes elliptiques. Ce chapitre développe certains éléments mathématiques ainsi que leur implantation en AMNS.

Dans les années 80, Miller [Mil97] et Koblitz [Kob87] ont proposé l'utilisation des courbes elliptiques dans la cryptographie. Bien qu'au début cette méthode n'ait pas trouvé de succès dans le monde industriel [NSA08], elle s'est imposée par ses avantages dans le monde de la recherche académique.

L'avantage majeur de l'utilisation des courbes elliptiques dans la cryptographie est qu'elle nécessite des clés publiques de taille plus petite que celles employées par les cryptosystèmes tels que le RSA [RSA78b] pour un même niveau de sécurité. Par exemple, la taille du clé pour une niveau de sécurité 128 bits dans ECC est de 256 bits, tandis que pour RSA la taille de la clé est de 3072 bits.

Dans ce chapitre nous proposons, une implantation en AMNS des opérations arithmétiques sur les courbes elliptiques. Notre contribution peut être résumée comme suit : en utilisant notre architecture matérielle de la multiplication modulaire dans le système de représentation AMNS, nous proposons une conception pour les opérations ECC : l'addition de deux points et le doublement d'un point. Pour cela, nous avons utilisé l'architecture séquentielle de la multiplication modulaire dans le système de représentation AMNS afin d'avoir une implantation des opérations arithmétiques sur les courbes elliptiques optimisées.

Ce chapitre est organisé comme suit. Nous commençons par présenter dans la partie 5.1 l'état de l'art de l'arithmétique sur les courbes elliptiques. Ensuite, nous nous intéressons dans la partie 5.2.3 à la multiplication scalaire sur courbe elliptique. Nous proposons dans la partie 5.3 une architecture matérielle pour l'implantation de la multiplication scalaire sur les courbes elliptiques.

5.1 Les courbes elliptiques

Dans cette partie, certaines propriétés importantes des courbes elliptiques sont présentées, notamment l'équation algébrique à laquelle elles obéissent et le nombre de points qui se trouvent sur la courbe. Nous présentons en outre des algorithmes de multiplication scalaire.

5.1.1 Équation de Weiestrass

Une courbe elliptique E sur un corps $\mathbb{F}_p$ est définie par l'ensemble des couples $(x, y) \in E$, tels que :

$$E : y^2 = x^3 + ax + b,$$

où $a, b \in \mathbb{F}_p$.

À cette courbe est associé le discriminant, noté Δ, non nul, et défini comme suit :

$$\Delta = -16(4a^3 + 27b^2),$$

La condition de $\Delta \neq 0$ garantit que chaque point sur la courbe admet une seule tangente. Un point sur la courbe est représenté par le couple (x, y). Cette courbe est muni d'un point particulier nommé point à l'infini et noté $\mathcal{O}$. L'ensemble des points de la courbe E muni de l'opération d'addition, avec le point à l'infini $\mathcal{O}$, forment un groupe additif. Le point à l'infini joue le rôle de l'élément neutre de l'addition dans ce groupe.

5.1.2 Les courbes elliptiques et la cryptographie

La multiplication scalaire sert de base à la plus part des crypto protocoles basés sur les courbes elliptiques. Nous pouvons citer par exemple l'échange de clés de Diffie Hellman [DH76] et le chiffrement EC ElGamal [ElG85]. Le multiplication scalaire est l'opération $k.P$, où k est un entier et P un point d'une courbe elliptique E. Cette opération est définie comme suit :

$$Q = k.P = \underbrace{(P + P + \cdots + P)}_{k fois},$$

Q est donc un point de la courbe E.

5.2 Arithmétique des courbes elliptiques

Nous considérons dans toute la suite de ce chapitre une courbe elliptique E. L'arithmétique des courbes peut être représentée sur trois niveaux. Le niveau le plus bas est l'arithmétique modulaire. Celle-ci est utilisée pour l'implantation des opérations d'addition et de doublement de points dans l'ECC. L'addition et le doublement sont ensuite utilisés comme composants de base pour l'implantation de la multiplication scalaire. Ces dépendances sont représentées graphiquement sur la figure 5.1. Dans les parties 5.2.1 et 5.2.3 nous présentons le système de coordonnées projectif et affine ainsi que l'algorithme de multiplication scalaire.

5.2.1 Choix de système de coordonnées

Il existe plusieurs systèmes de cordonnées pour l'arithmétique sur les courbes elliptiques. Selon le cas d'usage, certains système peuvent être plus intéressants

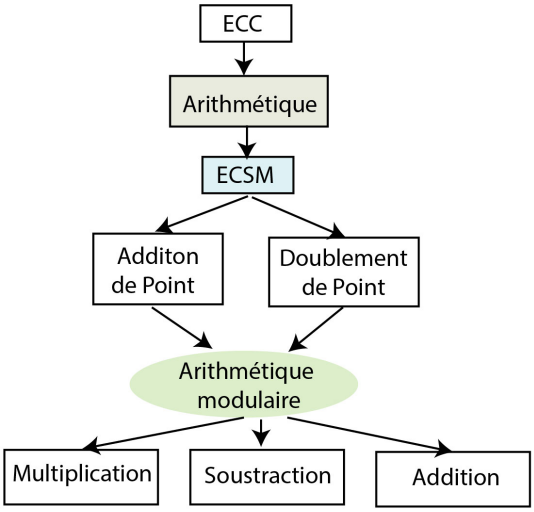

FIGURE 5.1 – Arithmétique des courbes elliptiques

que des autres. Dans cette section, nous représentons certains des systèmes les plus utilisés.

5.2.1.1 Cordonnées affines

Avant d'étudier tous les différents systèmes de coordonnées disponibles, il est nécessaire de comprendre le système de coordonnées affines [BSSS99]. Ce système de coordonnées est utilisé dans les cryptosystème à courbe elliptique étant donné sa nature compacte et le fait que ce système soit la base de tous les autres systèmes de coordonnées. Cependant, il est rarement employé dans les applications de bas niveau à cause de son coût élevé dû à des divisions.

On représente un point affine A à l'aide du couple (x_A, y_A) qui vérifie l'équation $y_A^2 = x_A^3 + ax_A + b \pmod{p}$.

Le point à l'infini de ce système, noté $\mathcal{O}$, sera représenté soit par point de coordonnées $(0, 0)$, soit par une valeur booléenne pour indiquer que le point n'est pas valide sur la courbe. Aussi, l'opposé d'un point affine $A = (x_A, y_A)$ est représenté par $-A = (x_A, -y_A)$. Soit $A = (x_A, y_A)$ et $B = (x_B, y_B)$ deux points de la courbe E dans $\mathbb{F}_p$. L'addition de deux points de la courbe E en coordonnées affines se fait comme suit :

On a $R(x_3, y_3) = P + Q$, avec $x_3 = \lambda^2 - x_1 - x_2$ et $y_3 = \lambda(x_1 - x_3) - y_1$ où $\lambda = \frac{(y_2 - y_1)}{x_2 - x_1}$ si $P \neq Q$ et $\lambda = \frac{(3x_1^2 + a)}{2y_1}$ si $P = Q$

5.2.1.2 Les Cordonnées Jacobienne

Les coordonnées jacobiennes sont également bien connues, bien qu'elles soient parfois appelées coordonnées projectives ou coordonnées homogènes [BSSS99]. Ce système de coordonnées représentent un point comme (X, Y, Z) où X, Y et Z satisfont l'équation : $Y^2Z = X^3 + aXZ^2 + bZ^3 (\text{mod } p)$ avec a et b sont les mêmes paramètres de courbe que pour les coordonnées affines. Nous représentons le point à l'infini du système de coordonnées jacobiennes par O par $(1 : 1 : 0)$.

Un point projectif peut être converti en un point affine à l'aide des équations :

— $x_A = XZ^{-1} (\text{mod } p)$

— $y_A = YZ^{-1} (\text{mod } p)$

Il est évident, à partir de ce système d'équation, qu'un point affine peut être converti en plusieurs représentations projectives différentes et inversement. En fait, deux points projectives (X_1, Y_1, Z_1) et (X_2, Y_2, Z_2) se confondent si $X_1Z_2 = X_2Z_1$ et $Y_1Z_2 = Y_2Z_1$. Il est également issu de la définition de l'opposé d'un point affine et des équivalences ci-dessus que l'opposé d'un point projectif $P = (X, Y, Z)$ est le point $-P = (X, -Y, Z)$.

5.2.1.3 Les méthodes ZADDU et ZADDC

Dans la suite de ce chapitre, seul le système de coordonnées Jacobien sera considéré. Dans [Mel07], Meloni a proposé une nouvelle formule pour accélérer l'addition des points dans le système de coordonnées Jacobien, avec la condition que les deux points partagent la même coordonnée Z.

Avec l'approche de Meloni l'addition de deux points ayant la même coordonnée Z coûte $5M + 2S$ alors que l'addition classique coûte $11M + 5S$, où M désigne une multiplication modulaire et S un carré modulaire.
Soit $P_1 = (X_1, Y_1, Z)$ et $P_2 = (X_2, Y_2, Z)$ deux points, où $P_1 \neq P_2$. La somme de ces deux points $P_1 + P_2 = P_3 = (X_3, Y_3, Z_3)$ avec la méthode ZADDU est

faite comme suit :

$$AlgorithmeZADDU \begin{cases} C \leftarrow (X_1 - X_2)^2 \\ W_1 \leftarrow X_1 C \\ W_2 \leftarrow X_2 C \\ Z_3 \leftarrow Z(X_1 - X_2) \\ D \leftarrow (Y_1 - Y_2)^2 \\ A_1 \leftarrow Y_1(W_1 - W_2) \\ X_3 \leftarrow DW1 - W2 \\ Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1 \\ R := (X_3, Y_3, Z_3) \\ S := (W_1, A_1, Z_3) \quad return(R = P + Q,\ S) \end{cases} \tag{5.1a}$$

Goundar et al [GJM10] propose une autre formule, d'addition de points, nommée ZADDC, ayant la même coordonnée Z. À partir des points P_1 et P_2, cette formule calcule le couple de points (R, S) tels que $R = P_1 + P_2$ et $S = P_1 - P_2$. Cette méthode coûte $6M + 3S$.

L'algorithme ZADDC est le suivant :

$$AlgorithmeZADDC \begin{cases} C \leftarrow (X1 - X2) \\ W_1 \leftarrow X_1 C \\ W_2 \leftarrow X_2 C \\ Z_3 \leftarrow Z(X_1 - X_2) \\ D_1 \leftarrow (Y_1 - Y_2)^2 \\ A_1 \leftarrow Y_1(W_1 - W_2) \\ X_3 \leftarrow D_1 - W_1 - W_2 \\ Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1 \\ D_2 \leftarrow (Y_1 + Y_2)^2 \\ X_4 \leftarrow D_2 - W_1 - W_2 \\ Y_4 \leftarrow (Y_1 + Y_2)(W_1 - X_4) - A_1 \\ R := (X_3, Y_3, Z_3) \\ S := (X_4, Y_4, Z_3) \quad return(R = P + Q, S = P - Q) \end{cases} \tag{5.2a}$$

L'algorithme de doublement, nommé DBLU, avec mise à jour de la coordonnée Z est le suivant :

$$AlgorithmeDBLU \begin{cases} B \leftarrow X^2 \\ E \leftarrow Y^2 \\ L \leftarrow E^2 \\ T \leftarrow 8L \\ N \leftarrow 2((X_1 + E)^2 BL) \\ M \leftarrow 3B + a \\ X_2 \leftarrow M^2 2N \\ Y_2 \leftarrow M(NX_2)T \\ Z_2 \leftarrow 2Y_1 \end{cases} \tag{5.3a}$$

5.2.2 Multiplication Scalaire sur ECC : principe et état de l'art

La multiplication scalaire est l'opération principale de la cryptographie sur les courbes elliptiques. Par exemple, la multiplication scalaire (ECSM) est utilisée dans les protocoles cryptographiques tel que l'ECDSA (Elliptic Curve Digital Signature Algorithm), l'ECDH (Elliptic Curve Diffie Hellman), l'EC-ELGAMAL (Elliptic Curve ElGamal).

Pour des raisons de sécurité, il est nécessaire que l'algorithme utilisé pour la multiplication scalaire soit régulier c'est à dire sans branchement conditionnel qui laisserait fuir des informations sur le scalaire utilisé. Dans la suite de cette thèse, nous ne considérons donc que des algorithmes réguliers. Il existe plusieurs méthodes et techniques pour implémenter la multiplication scalaire $k.P$ de manière efficace. Les algorithmes réguliers les plus utilisés sont : le Montgomery ladder, le Double-and-Add always, le Montgomery ladder avec les coordonnées Co-Z, qui sont détaillés respectivement dans les algorithmes 15, 16 et 17.

— **Le Montgomery ladder** est détaillé dans l'algorithme 15.

Algorithm 15 Montgomery Ladder[JY02]

Entrées: $P \in E(_q)$ avec $k = (k_{n-1}, \cdots, k_0)_2 \in \mathbb{N}$

Sorties: $\mathbf{Q = kP}$

1: $R_0 \leftarrow 0$; $R_1 \leftarrow P$
2: **pour** $i = (n-1) \ldots 0$ **faire**
3: $b \leftarrow k_i$; $R_{1-b} \leftarrow R_{1-b} + R_b$
4: $R_b \leftarrow 2R_b$
5: **fin pour**
6: retourner R_0

Chaque itération comprend une addition de points suivie d'un doublement de points. Cette méthode est sûre contre les attaques de type SPA ainsi que celle de type safe-error.

— **Le Double-and-Add always** est présenté dans l'algorithme 16.

Cette méthode est sûre contre les attaques de type SPA, mais elle est vulnérable aux attaques de type "safe-error".

— **Le Montgomery Ladder avec les coordonnées Co-Z** est détaillé dans l'algorithme 17. Cette méthode tire parti des méthodes ZADDU, ZADDC

Algorithm 16 Left-to-right double and add always [Cor99]

Entrées: $P \in E(_q)$ avec $k = (k_{n-1}, \cdots, k_0)_2 \in \mathbb{N}$

Sorties: **Q** $=$ **kP**

1: $R_0 \leftarrow P$; $R_1 \leftarrow 2P$
2: **pour** $i = (n-2) \ldots 0$ **faire**
3: $R_0 \leftarrow 2R_0$ $\mathrm{k}_i = 1$ $\mathrm{R}_0 \leftarrow R_0 + P$
4: **fin pour**
5: retourner R_0

et DBLU pour améliorer le coût du Montgomery Ladder classique. Ces coûts sont présentés dans la section 5.2.3.

Algorithm 17 Montgomery Ladder avec les cordonnées Co-Z [GJM10]

Entrées: $P \in E(_q)$ avec $k = (1, k_{n-1}, \cdots, k_0)_2 \in \mathbb{N}$

Sorties: **Q** $=$ **kP**

1: $(R_1, R_0) \leftarrow DBLU(P)$
2: **pour** $i = (n-2) \ldots 0$ **faire**
3: $(R_{1-k_i}, R_{k_i}) \leftarrow ZADDC(R_{k_i}, R_{1-k_i})$
4: $(R_{k_i}, R_{1-k_i}) \leftarrow ZADDU(R_{1-k_i}, R_{k_i})$
5: **fin pour**
6: retourner R_0

5.2.3 Analyse des performances

Dans le tableau 5.1, nous présentons le coût des certaines méthodes d'additions sur des courbes elliptiques.

Opération	Notation	Système	Coût
Addition basic coordonnée affine	ADD-A	$\mathcal{A} \rightarrow \mathcal{A}$	1I + 2M + 1S
Addition basic coordonnée Jacobienne	ADD-J	$\mathcal{J} \rightarrow \mathcal{J}$	12M + 4S
Addition dans le système de coordonnée Co-Z	ZADDU	$\mathcal{J} \rightarrow \mathcal{J}$	5M +2S

Le conjugué de l'addition dans le système de coordonnée Co-Z	ZADDC	$\mathcal{J} \rightarrow \mathcal{J}$	6M +3S

TABLE 5.1: Comparaison des performances des algorithmes d'addition.

Les symboles $\mathcal{A}$ et $\mathcal{J}$ correspondent respectivement aux coordonnées affines et Jacobiennes. nous notons M : Multiplication, S : Puissance (Square), I : Inversion.

Le tableau 5.2 présente les coûts des méthodes de multiplication scalaire présentées plus haut. Nous pouvons constater que l'algorithme 17 est le moins coûteux. C'est donc ce dernier que nous avons choisi d'implémenter sur FPGA.

Algorithme	Notation	Système	Coût
Double and ADD	ADD-A	$\mathcal{A} \rightarrow \mathcal{A}$	$n(1I+2M+1S)$
Montgomery Ladder	ADD-J	$\mathcal{J} \rightarrow \mathcal{J}$	$n(14M+10S)$
Montgomery Ladder avec Co-Z	ZADDU	$\mathcal{J} \rightarrow \mathcal{J}$	$n(9M+7S)$

TABLE 5.2 – Comparaison des performances des algorithmes d'addition, où n est la taille du scalaire k

5.3 Modélisation et Implantation en description RTL

Dans cette partie nous proposons une architecture matérielle qui suit les étapes décrite dans l'algorithme 17.

L'algorithme 17 utilise les algorithmes ZADDU, ZADDC et DBL afin d'effectuer les opérations de multiplication scalaire sur ECC.

Dans cette section, nous proposons une nouvelle architecture matérielle pour implémenter la multiplication scalaire dans le système de représentation AMNS. Dans la partie 5.3.1, nous présentons notre architecture séquentielle pour l'implantation de la multiplication scalaire sur ECC. Les résultats d'implantation seront présentés dans la partie 5.4.

5.3.1 Modélisation

Notre architecture matérielle de la multiplication scalaire dans le système de représentation AMNS est basée sur l'algorithme 18. Cet algorithme peut être développé comme suit :

Algorithm 18 Multiplication scalaire sur ECC - étapes principales

$$(R_1, R_0) \leftarrow \mathbf{DBLU}(P) \tag{5.4}$$
$$(R_{1-k_i}, R_{k_i}) \leftarrow \mathbf{ZADDC}(R_{k_i}, R_{1-k_i}) \tag{5.5}$$
$$(R_{k_i}, R_{1-k_i}) \leftarrow \mathbf{ZADDU}(R_{1-k_i}, R_{k_i}) \tag{5.6}$$

Nous rappelons que l'entrée P est un point sur la courbe elliptique et le scalaire K est un entier dans $\mathbb{N}$. Le point P est représenté dans le domaine de Montgomery. Nous proposons une conception de notre architecture dans la Fig. 5.2. Dans cette figure, l'opérateur *Multiplication scalaire sur ECC*, traite les étapes (5.4), (5.4) et (5.5) de l'algorithme de multiplication scalaire sur ECC dans l'algorithme 18.

La structure de notre architecture est donnée par la Fig. 5.2. Dans cette figure, l'opérateur *Multiplication scalaire sur ECC dans AMNS*, traite les étapes (5.1), (5.2)et (5.3) de l'algorithme de multiplication scalaire sur ECC dans l'algorithme 18. En premier lieu, il charge les cordonnées du point P dans le système CO-Z et le scalaire K, ainsi que le coefficient de chaque valeur d'entrée m'_j et m_j de la mémoire. Ensuite, il traite les résultats intermédiaires R_0, R_1, R_{k_i}, R_{1+k_i} et les stocke dans des registres locaux dédiés. Notamment, cela permet d'utiliser R_0, R_1, R_{k_i}, R_{1+k_i} dans l'opérateur *Multiplication scalaire sur ECC dans AMNS* comme valeurs préliminaires pour calculer : $1 \times P$, $2 \times P$, $3 \times P, \ldots$ $\mathrm{k}_n \times P$. Ainsi, le registre dédié à S fournit le résultat de KP à la mémoire.

Le comportement du flux de données de l'algorithme de multiplication scalaire peut être programmé de différentes manières pour tirer parti du parallélisme en tenant compte des contraintes de ressources. Nous avons utilisé une approche modulaire, qui est divisée en six unités principales : Contrôleur, Mémoire, Multiplication Modulaire dans AMNS (MMA), **ZADDC**, **ZADDU** et **DBL**.

Le module de doublement **DBL** effectue l'étape 5.4 de l'algorithme de multiplication scalaire sur ECC. Précisément, il s'agit du système d'équations, qui est décrit en détail dans l'algorithme 5.3. Les résultats intermédiaires B, E, L, T, N, M, X_2, Y_2 et Z_2 seront stockés dans des registres locaux dédiés. En particulier, cela permet d'utiliser les variables X_2, Y_2 et Z_2 comme entrées dans

le module **ZADDU** dans la boucle suivante. Cependant, il faut noter que le module de doublement utilise le module **MMA** pour gérer toutes les opérations de multiplication modulaire.

La deuxième étape consiste à traiter les équations dans les algorithmes 5.1 et 5.2. Ces fonctions sont exécutées par les modules **ZADDU** et **ZADDC**. En effet, la méthode utilisée pour calculer la multiplication de points est basée sur la décomposition binaire du scalaire K.

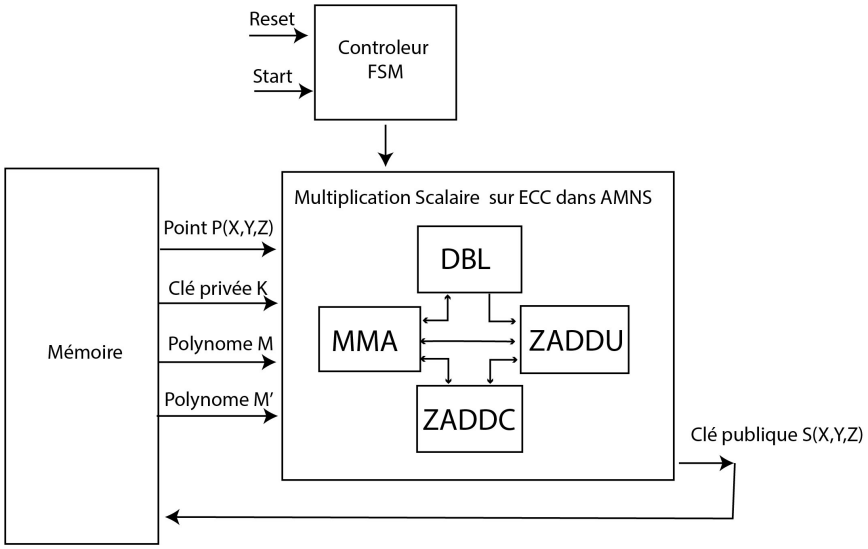

FIGURE 5.2 – Architecture de la multiplication scalaire

5.3.2 Implantation Matérielle

Dans cette partie, nous présentons une architecture matérielle qui peut être utilisée pour calculer une multiplication scalaire sur ECC dans le système de représentation AMNS.

Cette architecture est composée de deux étages. Le premier étage, qui est le doublement d'un point, prend 10 étapes pour s'exécuter. Dont chaque étape peut être exécute en plus qu'un cycle. le deuxième étage qui regroupe le deux opérations ZADDU et ZADDC, qui est effectuée respectivement en 9 étapes et 10 étapes. Le nombre de cycles d'horloge de DBLU, ZADDU et ZADDC dépend des opérations d'arithmétique modulaire dans AMNS. Par conséquent, si la multiplication utilise 24 cycles d'horloge et que la soustraction utilise 2 cycles d'horloge, alors 24 cycles d'horloge prendront des étapes qui incluent la multiplication et la soustraction.

L'architecture modulaire de la multiplication scalaire ECSM est conçue à

l'aide d'un fonctionnement séquentiel des différentes unités. Dans les parties 5.3.2.2, 5.3.2.4 et 5.3.2.3, nous avons démontré ce fait, tout en présentant nos architectures séquentielle pour les modules DBL, ZADDU et ZADDC.

5.3.2.1 Arithmétique Modulaire (AM)

Les opérations décrites dans les parties 5.3.2.2, 5.3.2.4 et 5.3.2.3, nécessitent des fonctions arithmétiques modulaires. Le circuit d'addition (**Add**) et de soustraction (**Sub**), de multiplication de coefficient (**MC**) et de multiplication modulaire(**M**) sont généré dans le système de représentation AMNS.

Pour les circuits d'addition (**Add**) et de soustraction (**Sub**), l'opération d'addition et soustraction modulaire est une simple addition et soustraction polynomiale, suivie d'une réduction interne. Cette opération s'exécute en deux cycles d'horloge. Pour le circuit **MC**, est une multiplication simple des coefficients suivie d'une réduction interne. Ce circuit effectue une multiplication des coefficients en deux cycles d'horloge.

Le circuit de multiplication modulaire, c'est l'IP matérielle séquentielle de la multiplication modulaire dans le système de représentation AMNS, qui est décrit dans la partie 4.2.2.1.

5.3.2.2 DBL

Le premier étage de l'algorithme 18 consiste à calculer trois valeurs X_2, Y_2, Z_2. Ces valeurs sont calculées à partir des équations 5.3. Ces valeurs représentent les entrées de la deuxième étape de l'algorithme 18.

Tout d'abord, nous utilisons les modules **M**, **MC**, **Add** et **Sub** pour trouver les variables intermédiaires M, T, L, N et E. Le résultat de cette étape est stocké dans des registres de la mémoire BRAM, afin d'être utilisé dans l'étape suivante.

Comme le montre la figure 5.3, le nombre de cycles d'horloge requis pour calculer le module DBL dépend du nombre de cycles d'horloge requis pour calculer le multiplication modulaire, la soustraction, l'addition. Elle nécessitent respectivement 24, 2 et 2 cycles d'horloge pour effectuer ses calculs. Le module DBLU est construit sur la base des opérations arithmétique semi-parallèle, ces opérations arithmétiques séquentielles ont besoin de plus qu'un cycle d'horloge pour fonctionner. Dans la première étape, nous utilisons deux opérations de multiplication **M**. Les entrées utilisées sont les coordonnées $P(X_1, Y_1)$ du point de la courbe elliptique. La taille de chaque entrée est de 256 bits, stockés dans 8 coefficients $w = 36$ bits. Les sorties sont enregistrées dans les registres B, E. Qui seront utilisée en entrée des modules **MC** et **Add** dans la deuxième étape.

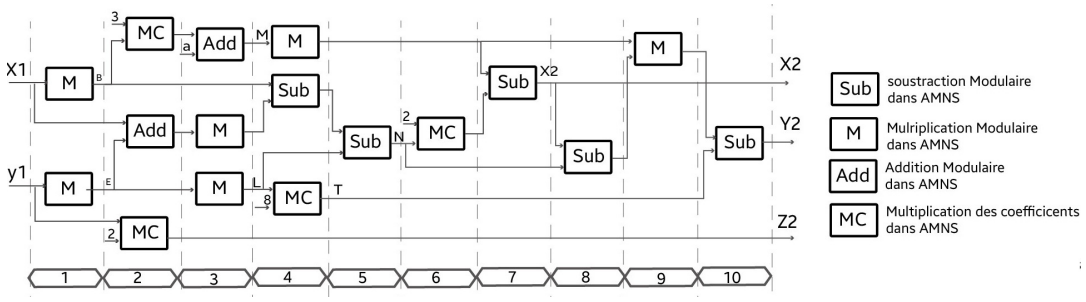

FIGURE 5.3 – Architecture du module DBL

Dans la deuxième étape, nous utilisons deux multiplications du coefficient **MC**, leurs entrées sont les variables B et Y_1. Ces deux opérations permettent de calculer respectivement les fonctions $3B$ et $2Y_1$. Dans la même étape, nous utilisons l'opération d'addition modulaire **Add**, en prenant les variables X_1 et E en entrée pour calculer la fonction X_1+E. Ces trois opérations sont exécutées en 2 cycles d'horloge. Les sorties seront enregistrées dans les registres Z_2, M, N_1 qui seront utilisée en entrée à l'étape suivante.

L'étape 3, contient deux opérations de multiplication modulaire **M** et une opération d'addition **Add**. Comme nous l'avons déjà dit, l'opération **M** nécessite 24 cycles, tandis que l'opération **Add** nécessite 2 cycles. Cependant, ce n'est qu'après avoir terminé ces trois opérations (24 cycles d'horloge) que la transition vers l'étape suivante peut être effectuée.

Ce principe s'applique à toutes les autres étapes qui suivent. 24 cycles d'horloge pour exécuter les étapes 4 et 9 et pour chacune des étapes suivantes $5, 6, 7, 8$ et $10, 2$ cycles d'horloge. Alors le calcul de DBL nécessite 108 cycles d'horloge.

5.3.2.3 ZADDC

Le deuxième étage de l'algorithme 18 est composée de deux opérations, qui regroupent le calcul de l'addition de deux points ZADDU et l'addition conjuguée ZADDC dans le système de coordonnée Co-Z. Le circuit proposé dans la figure 5.2, pour calculer le équations 5.5 et 5.6 de l'algorithme 18 nécessite six registres $(R_x, R_y, R_{1x}, R_{2y}, Z_x, Z_y)$ qui sont utilisés $(n-1)$ fois pour stocker les cordonnées du point (R_0, R_1). Les six registres sont réutilisés dans chaque itération dans l'opération ZADDC et ZADDU comme indiqué dans l'algorithme 18.

Dans cette partie nous détaillons l'architecture du module ZADDC. Ce module consiste à calculer cinq valeurs X_4, Y_4, X_3, Y_3 et Z_4. Ces valeurs sont calculées à partir des équations 5.2. Ces variables sont stockées dans des registres dédiées de la mémoire BRAM. Afin d'être réutilisable à chaque itération

de l'algorithme 18. L'architecture proposée sur la figure 5.4 est implémenté en 10 étapes. Six des étapes utilisent le module **M** de multiplication modulaire, qui nécessite 24 cycles d'horloge. Les quatre autres étapes utilisent le module de soustraction **Sub**, qui se produit en 2 cycles d'horloge. Ce qui nous donne un total de 152 cycles. Cette architecture est conçue en utilisant des opérations séquentielle pour optimiser le nombre de cycles d'horloge.

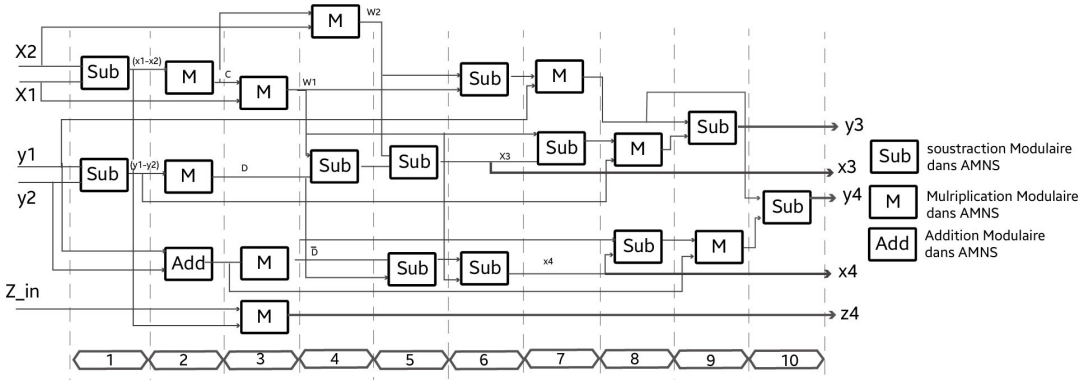

FIGURE 5.4 – Architecture du module ZADDC

5.3.2.4 ZADDU

La figure 5.5 illustre l'architecture matérielle de l'opération ZADDU de l'algorithme 18. Cette architecture est réalisée sur les 9 étapes. Chaque étape exécute une ou plusieurs équations du système d'équations présenté dans 5.1. Le module ZADDU consiste à donner en sortie cinq variables X_4, Y_4, X_3, Y_3 et Z_4. Ces variables sont stockées dans des registres dédiés de la mémoire BRAM. Afin d'être réutilisable à chaque itération de l'algorithme 18. L'architecture exécute

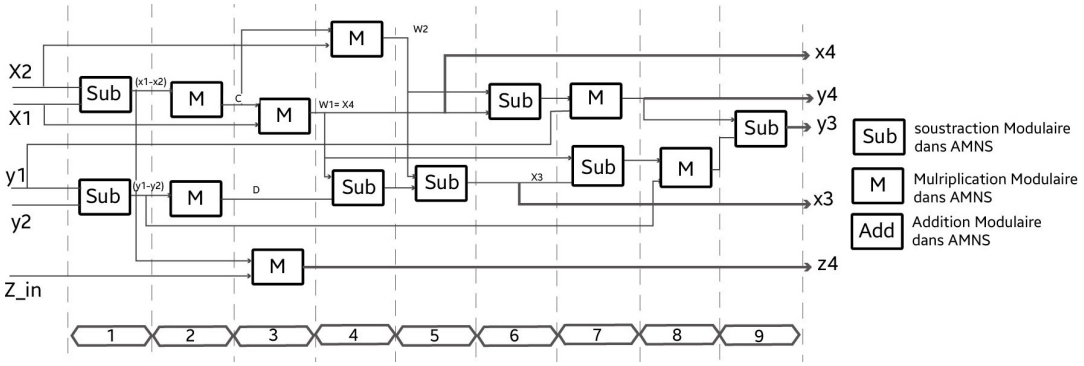

FIGURE 5.5 – Architecture du module ZADDU

les opération dans le système d'équations (5.1a) d'une façon séquentielle. Les

modules que nous avons utilisés dans cette architecture sont : **Sub**, **M** et **Add**. L'implantation du module **Sub** prends 2 cycles, le module **M** s'exécute en 24 cycles, le module **Add** nécessite 2 cycles. Ce qui nous donne un total de 128 cycles.

5.3.2.5 Le contrôleur

La partie contrôleur comprend une FSM, un ensemble d'instructions, détaillés dans les parties 5.3.2.2, 5.3.2.3 et 5.3.2.4, stockées dans la ROM et des registres pour stocker et exploiter les variables '**Start**' et '**Reset**'. En utilisant cette méthode, seules des modifications mineure de la FSM et des instructions d'algorithmes cible sont nécessaires pour générer l'ECPM.

Pour $w = 36$ et un module p de 256 bits, la FSM contrôlant notre architecture est illustré dans la Fig. 5.6. elle est composée de 32 états, l'état 1 étant l'état de départ. Le module DBL de l'algorithme principal commence par l'état 1 et se termine par l'état 10.

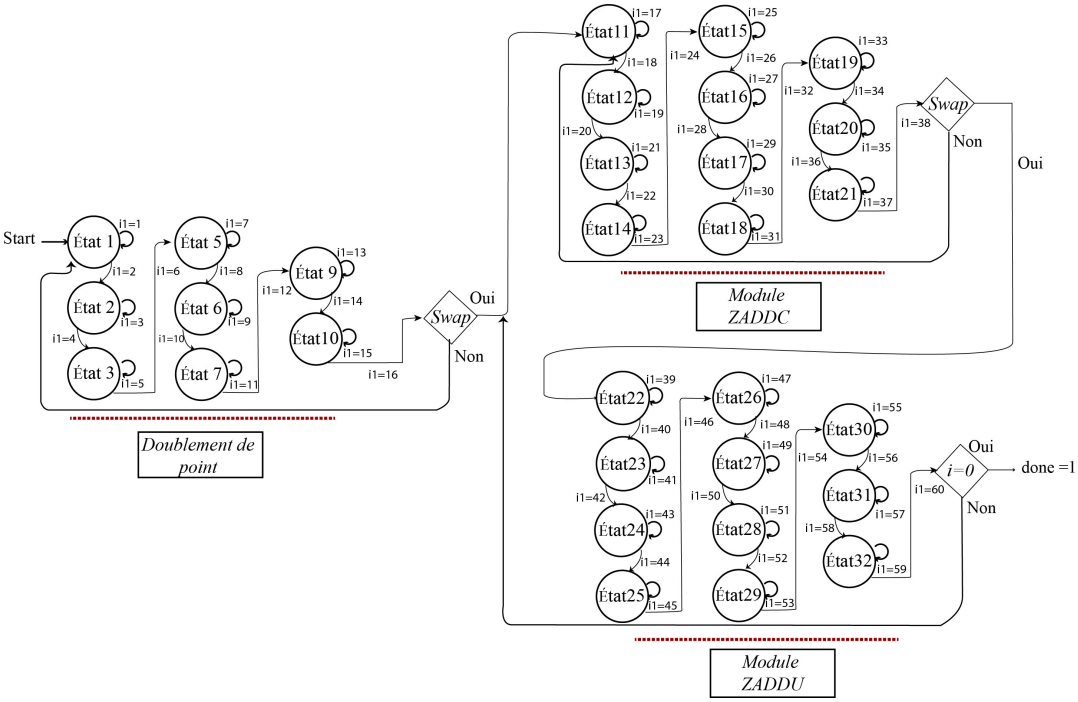

FIGURE 5.6 – Machine à état finis de la multiplication scalaire de ECC

À l'état 10, il faut vérifier si le processus du module DBL est terminé. Si c'est "*Oui*", alors FSM passe au module ZADDC, si c'est "*Non*", alors FSM revient au point de départ. Le passage vers le module ZADDC est exécuté dans un sous-programme, qui commence à l'état 11 et se termine à l'état 21. Un signal

$done\_zaddc = 1$ est émis pour indiquer que le processus ZADDC est terminé et que les le résultats sont prêt à être utilisé par le sous-programme suivant. À l'état 21, la condition est faite pour vérifier la valeur du bit $done\_zaddc$. Si c'est égal à 1, alors la FSM passe au module ZADDU. Si c'est 0, alors la FSM revient à l'état 11. Le module ZADDU est exécuté dans un sous-programme, qui commence à l'état 22 et se termine à l'état 32.

Dans l'état 32, la condition est mise afin de savoir si le module ZADDU est terminé. Si c'est "Oui", alors la FSM transmis un signal "$done$" pour indiquer que le processus de multiplication scalaire est terminé et que les coordonnées du point kP sont obtenues. Si c'est "Non", alors la FSM retourne à l'état 11 pour le k_i suivant.

Une fois que i est égal à zéro, les résultats sont prêts et le signal "$done$" est mis à 1 pour indiquer la fin de la multiplication.

5.4 Résultats d'implantation

Nous avons complètement implémenté et validé l'architecture séquentielle pour l'algorithme DBLU et ZADDU et ZADDC sur FPGA. Les résultats sont donnés pour 2 FPGAs : Virtex 7 (xc7vx1140tffg 1932 -2) et virtex ultrascale (xcvu440-flga 2892-3-e). Nous utilisons l'outil Vivado 2020.1 pour la synthèse.

Ensuite, nous présentons une estimation théorique pour une implantation de la multiplication scalaire sur ECC. Les résultats que nous montrons dans le tableau 5.3 de l'ECC sont basé sur les résultats d'implantation des algorithmes DBLU, ZADDU et ZADDC qui constituent la multiplication scalaire sur ECC. En fait, nous avons utilisés la formule suivante : $speed\ ECC = speed\ DBLU + (speed\ ZADDU + speed\ ZADDC) \times 254$. Cette formule est délivrée partir de l'algorithme 17, en effet la multiplication par 254 n'est que la valeur de $n-2$, avec $n-2$ représente le nombre d'itérations à faire afin de calculer la multiplication scalaire sur ECC. D'autre part, nous avons calculer la valeur de la fréquence comme suit : $Fréq.ECC = \frac{nombre\ de\ cycle}{speedECC}$, avec $nombre\ de\ cycle = nombre\ de\ cycle\ de\ DBLU + (nombre\ de\ cycle\ de\ ZADDU + nombre\ de\ cycle\ de\ ZADDC) \times 254$, l'application numérique nous donne : $nombre\ de\ cycle\ ECC = 108 + (128 + 152) \times 254 = 71228$ cycles.

Famille	Module	LUT	DSP	Fréq. (Mhz)	speed (μs)	#cycle
Virtex Ultrascale	ZADDU	11849	504	115.06	1.112	128
	ZADDC	19633	1002	63.17	2.406	152
	DBLU	10003	422	154.34	0.699	108
	ECSM	16577	828	79.02*	901.307*	71228
Xilinx V7	ZADDU	11689	510	74.31	1.722	128
	ZADDC	17864	854	207.29	0.733	152
	DBLU	8022	326	152.95	0.706	108
	ECSM	267444	1806	113.20*	629.186*	71228

TABLE 5.3: Résultat théorique d'implantation matérielle des trois modules DBLU, ZADDC et ZADDU pour p=256 et w=64.

L'analyse des résultats d'implantation de notre architecture montre que le temps d'exécution de l'algorithme ZADDU sur le Virtex ultrascale est plus rapide de 1.54 fois de celui implanter sur le FPGA Xilinx V7. Tandis que le temps d'implantation de l'algorithme ZADDC sur Virtex ultrascale est plus lent de 3.28 fois de celui-ci de Xilinx V7. Néanmoins, le temps d'implantation de l'algorithme DBLU sur Virtex ultrascale est plus rapide de 1.01 fois de celui-ci de la Xilinx virtex 7.

En conséquence, le nombre des LUTs utilisées sur virtex Ultrascale pour l'implantation des algorithmes ZADDU, ZADDC et DBLU est plus grande de 1.02 au 1.24 fois de celui utilisé sur Xilinx V7.

Ce qui concerne le temps estimé de l'implantation de l'ECC sur la virtex ultrascale est plus lent de 1.43 fois de celui de Xilinx Virtex 7. Ce qui explique la consommation de moins de ressource pour l'implantation de l'ECC sur la virtex ultrascale que celui de Xilinx V7.

5.5 Conclusion et perspectives

Dans ce chapitre, nous présentons la première conception matérielle pour les trois modules (DBL, ZADDU, ZADDC) constituant la multiplication scalaire sur ECC, dans le système de représentations AMNS. L'architecture que nous avons conçue intègre l'IP matérielle séquentielle de la multiplication modulaire que nous avons conçu au chapitre 3. Ainsi que l'addition modulaire et la sous-

traction modulaire dans l'AMNS. Les résultats que nous avons présentés nous encouragent à terminer l'implantation de la multiplication scalaire sur l'ECC, d'une part. D'autre part, nous donne l'avantage d'implanter un cryptosystème en AMNS sur FPGA.

Conclusion

Cette thèse, se concentre sur la conception des architectures efficaces pour réaliser une implantation matérielle rapide et robuste des arithmétiques modulaires et le calcul de la multiplication scalaire sur les courbes elliptiques dans le système de représentation AMNS. Nous proposons la première architecture matérielle conçue pour utiliser ce système de représentation.

Dans ces dix dernières années, la combinaison de l'AMNS et de la cryptographie asymétrique est devenue la source de plusieurs résultats de recherche. Le choix de l'AMNS apporte des avantages évidents au chiffrement asymétrique.

Durant cette thèse, nous avons conçu et proposé différentes architectures matérielles flexibles pour le calcul de l'arithmétique modulaire en AMNS. Dans le but d'intégrer ces architectures dans l'implantation matérielle de la multiplication scalaire sur ECC. Ensuite, nous avons pu valider et implanter nos architectures sur FPGA. Tout au long de cette thèse, nous avons prouvé que nos architectures ont des temps d'exécution plus rapide que ceux présentés dans l'état de l'art ce qui rend nos résultats compétitifs.

Dans le chapitre 2, nous présentons la première contribution de la thèse, qui est de concevoir une IP matérielle pour la multiplication modulaire de grands nombres dans le système AMNS. La multiplication modulaire est l'opération la plus coûteuse dans les calcul cryptographique dans les corps finis $\mathbb{F}_p$. L'optimisation de ces opération entraîne une accélération matérielles des primitives cryptographique sur les FPGA. Nous avons implantés la multiplication modulaire pour des différentes niveaux de sécurité $128, 256, 512$ bits en utilisant différents mots $w = 32, 36, 54, 64$. L'outil utilisé est Vivado HLS est un outil de conception haut niveau. Nous avons utilisé des directives d'optimisation afin d'améliorer les performances de l'implantation. Ce travail est présenté à la Conférence international CRISIS'19 [CDD+19]. Dans cet article, nous présentons les résultats d'une implémentation efficace (HW / SW) de l'algorithme de multiplication modulaire AMNS, qui peut être utilisé pour implémenter des primitives cryptographiques sur la courbe elliptique sur $\mathbb{F}_p$. Nous fournissons

des opérateurs arithmétiques et une architecture matérielle optimisés pour les grands entiers. Nous comparons également les résultats avec les implantations de la méthode CIOS pour la multiplication modulaire.

Dans le chapitre 3, nous présentons la deuxième contribution de cette thèse. Nous avons conçu deux architecture matérielle décrit en HDL. Ces sont les premiers architectures matérielles conçu pour le système de représentation AMNS. Nous avons utilisé l'outil Vivado design suite. Ce travail a été accepté dans une revue internationale dans le journal Elseiver JISA [CDD+21]. Dans cet article nous proposons deux architectures matérielles qui utilisent le système de numérotation modulaire adaptative (AMNS) pour effectuer une multiplication modulaire sur le corps fini $_p$. Nous proposons deux architectures matérielles : l'architecture semi-parallèle et l'architecture séquentielle. Notre architecture semi-parallèle offre l'implantation la plus rapide. La comparaison de nos résultats avec les dernières technologies met en évidence que nous fournissons l'implantation de multiplication modulaire la plus rapide sur les corps finis avec le débit le plus élevé.

La troisième contribution de cette thèse est présenté dans le chapitre 4. Nous nous intéressons à la protection de la multiplication scalaire sur la courbe elliptique contre les attaques de type DPA. À cette fin, nous avons adapté les travaux du chapitre 3 pour concevoir une architecture matérielle pour la multiplication modulaire randomisé de l'AMNS. Nous avons implémenté notre architecture sur FPGA. Une étude comparative entre nos résultats et l'état de l'art montre que nous avons l'implantation la plus rapide sur les corps finis.

Enfin, le quatrième et dernier aspect de cet article concerne les courbes elliptiques. Nous avons intégré le travail effectué au chapitre 3 afin de concevoir une architecture matérielle dédiée à la multiplication scalaire sur une courbe elliptique utilisant le système de coordonnées Co-Z.

Pour les travaux futurs, nous espérons dans un premier temps optimiser l'architecture semi-parallèle pour réduire la surface. Nous voulons utiliser l'algorithme modulaire dans AMNS pour implémenter en codesign les primitives cryptographique sur les courbes elliptique sur les corps finis. Dans un deuxième temps, nous étudions la sécurité matérielle de la multiplication scalaire sur l'ECC contre les différentes technique des attaques.

A Compléments AMNS

Sommaire

A.1 Exemples d'AMNS pour différents nombres premiers

Dans cette partie, nous présentons des exemples des AMNS que nous avons implantés dans les FPGAs, dont nous avons présenté leurs résultats d'implantation dans les chapitres $2, 3, 4 et 5$.

A.1.1 AMNS 1 : nombre premier de 128 bits

- $p = 201108062637595729734197794662548046727$
- $n = 7$
- $w = 32$
- $\rho = 2^{25}$
- $\gamma = 702362187074035873232048756042112947 14$

- $E(X) = X^7 - 2$
- $M(X) = 982334X^6 + 629671X^5 + 870460X^4 + 941733X^3 + 1419379X^2 + 773531X + 28053$
- $M'(X) = 1628998522X^6 + 2373754906X^5 + 585170709X^4 + 1759040465X^3 + 2683527316X^2 + 2221042308X + 1633367663$

A.1.2 AMNS 2 : nombre premier de 256 bits

- $p = 90625167144997146313007430523377040135861388266225474166759963586345305308073 09$
- $n = 4$
- $w = 64$
- $rho = 2^{55}$
- $\gamma = 5449513566037597494732743904035552484118157695860310196102572264854554630948$
- $E(X) = X^4 - 2$
- $M(X) = 1350934316191595X^4 + 306559457019683X^3 1462942292505637X^2 + 1071675124402294X + 125347164713036$
- $M'(X) = 4892364901222836083X^4 + 690495429144917824 3X^3 + 160507435803762929 4X^2 + 1045480459873565985X + 4583353295946333871$

A.1.3 AMNS 3 : nombre premier de 256 bits

- $p = 89248193021131078597118547412539930861853232536634595608325707428331368967297$
- $n = 7$
- $w = 36$
- $\rho = 2^{35}$
- $\gamma = 847127051188830185579652551246784710921740688093479456751225518431907390142 52$
- $E(X) = X^7 + 1$
- $M(X) = 1906720567X^7 + 1215374707X^6 837325398X^5 + 2035806901X^5 + 1557489541X^4 + 1907696928X^3 + 1423126491X^2 + 1368523256L$
- $M'(X) = 478004579418X^7 + 490974298953X^6 + 197983106937X^5 + 252637481467X^4 + 58719255200X^3 + 455094803966X^2 + 519302850232X + 243861089666$

A.1.4 AMNS 4 : nombre premier de 512 bits

- $p = 702874863424594515447692550137307332227889227292559579869341408646981403874670078045449204600274418873798291505783591363320257234194059786767834742605450 3$
- $n = 9$
- $w = 64$
- $rho = 2^{56}$
- $\gamma = 681662398265809125407117174635628246553788450277059817072598901933819739564562852910439387999058389808943497343483231108587057628700870095450332607853698 0$
- $E(X) = X^9 + 4$
- $M(X) = 104748093457737X^9$+$646942248450823X^8$+$269661006884787X^7$+ $473859440345830X^6 + 947027862533646X^5 + 420478013269129X^4$+ $874155495226884X^3$+$1327251688143931X^2$+$1044227364783912X$+314404311180720
- $M'(X) = 1684570584525241093X^9 + 8921605967769802537X^8$+ $6396306011105482628X^7$+$1113442470735529828 3X^6$+$1785435450278237808 5X^5$+ $1068792496654084049 1X^4$+$1544881738752155929 9X^3$+$3014092611686664 75X^2$+ $4885225144926015372X + 11972849539302326940$

A.1.5 AMNS 5 : nombre premier de 1024 bits

- $p = 1052330832709856649489060117151437466447616323677741512115892249956758370473017415875077035110898679586776582190168673004042569528089019726563663988193503648992182813184138109881869826826230787656414893849636677918071208727315988076552784367205773463912051254688830704427853511243339580516893816083094237294 59$
- $n = 20$
- $w = 64$
- $\rho = 2^{55}$
- $\gamma = 4122469844272815508451825181352250334615018355059252361675669332553804076238335118269742547868245452498045065428446793061670461995124949553794240940000371178689650685589536088164938911064977423695393558186657798289069033575683237586072652107584220803352534537960376565874787170748416352808440826546572248216 7$ E(X) = X^{2}0 − 1

- $M(X) = 64703455549927X^{20}+61623151120597X^{19}+169567785024896X^{18}+274520508351795X^{17} + 208455842638191X^{16} + 145330239636284X^{15}+32714434126729X^{14} + 30899504020284X^{13} + 222912920262998X^{12}+225605224819863X^{11} + 206143487422996X^{10} + 175156456458316X^{9}+229584502983000X^{8} + 240361729125285X^{7} + 266017830069949X^{6}+233806502373801X^{5} + 55683096217536X^{4} + 35968238735285X^{3}+10639603027118X^{2} + 74262660520133X + 14441179755148$
- $M'(X) = 1155142474573408963X^20 + 14788204276963147241X^{19}+9976103076307816655X^{18}+11764540663026940146X^{17}+82505551160891182 14X^{16}+2044989950061472970X^{15}+460119758899515961 2X^{14}+55518150051874000 33X^{13}+813530344641009040 4X^{12}+152048303592861804 01X^{11}+134870474189633280 85X^{10}+239223367907070769 5X^{9}+165694501567013454 90X^{8}+89743854386188775 06X^{7}+752518733755704208 2X^{6}+193880033028493249X^{5}+17673917148337498533X^{4}+641340129737399213 2X^{3}+203665404323018292 7X^{2}+142205821951789295 14X+760102559513122560$

A.1.6 AMNS Randomisé 6 : nombre premier de 256 bits

- $p = 90625167144997146313007430523377040135861388266225474166759963580345530807309$
- $n = 5$
- $w = 64$
- $\rho = 2^{48}$
- $\gamma = 5449513566037597494732743904035552484118157695860310196102572264854555463094 8$
- $E(X) = X^4 - 2$
- $M(X) = 14477579451936X^5 + 14477579451936X^4 + 3417025446624X^3 + 3417025446624X^2 + 14477579451936X + 958314944803$
- $M'(X) = 480148753499257229 1X^5 + 480148753499257229 1X^4+480148753499257229 1X^3+480148753499257229 1X^2+480148753499257229 1X+480148753499257291$

Bibliographie

[Atr65] AJ Atrubin. A one-dimensional real-time iterative multiplier. *IEEE Transactions on Electronic Computers*, (3) :394–399, 1965.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[BB79] GR Blakley and I Borosh. Rivest-shamir-adleman public key cryptosystems do not always conceal messages. *Computers & mathematics with applications*, 5(3) :169–178, 1979.

[BDK98] J-C Bajard, L-S Didier, and Peter Kornerup. An rns montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7) :766–776, 1998.

[BIP04] J.-C. Bajard, Laurent Imbert, and Thomas Plantard. Modular number systems : Beyond the mersenne family. In *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada*, pages 159–169, 2004.

[BIP05] J.-C. Bajard, Laurent Imbert, and Thomas Plantard. Arithmetic operations in the polynomial modular number system. In *17th IEEE Symposium on Computer Arithmetic (ARITH-17) 2005, Cape Cod, MA, USA*, pages 206–213, 2005.
Extended (complete) version available at : https://hal-lirmm.ccsd.cnrs.fr/lirmm-00109201/document.

[BM14] Jean-Claude Bajard and Nabil Merkiche. Double level montgomery cox-rower architecture, new bounds. In *International Conference on Smart Card Research and Advanced Applications*, pages 139–153. Springer, 2014.

[BÖPV03] Lejla Batina, Sıddıka Berna Örs, Bart Preneel, and Joos Vandewalle. Hardware architectures for public key cryptography. *Integration*, 34(1-2) :1–64, 2003.

[BP01] Thomas Blum and Christof Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE transactions on computers*, 50(7) :759–764, 2001.

[BSSS99] Ian Blake, Gerald Seroussi, Gadiel Seroussi, and Nigel Smart. *Elliptic curves in cryptography*, volume 265. Cambridge university press, 1999.

[BT15] Karim Bigou and Arnaud Tisserand. Single base modular multiplication for efficient hardware rns implementations of ecc. In *CHES : 17th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 9293, pages 123–140. Springer, 2015.

[CDD+19] Asma Chaouch, Yssouf Fangan Dosso, Laurent-Stéphane Didier, Nadia El Mrabet, Bouraoui Ouni, and Belgacem Bouallegue. Hardware optimization on fpga for the modular multiplication in the amns representation. In *International Conference on Risks and Security of Internet and Systems*, pages 113–127. Springer, 2019.

[CDD+21] Asma Chaouch, Laurent-Stéphane Didier, Fangan Yssouf Dosso, Nadia El Mrabet, Belgacem Bouallegue, and Bouraoui Ouni. Two hardware implementations for modular multiplication in the amns : Sequential and semi-parallel. *Journal of Information Security and Applications*, 58 :102770, 2021.

[CEES14] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. *The Zynq Book : Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International workshop on cryptographic hardware and embedded systems*, pages 292–302. Springer, 1999.

[Cra92] Richard E Crandall. Method and apparatus for public key exchange in a cryptographic system, October 27 1992. US Patent 5,159,632.

[Cur99] NIST Curves. Recommended elliptic curves for federal government use, 1999.

[DDEM+19a] Laurent-Stéphane Didier, Fangan-Yssouf Dosso, Nadia El Mrabet, Jérémy Marrez, and Pascal Véron. Randomization of Arithmetic over Polynomial Modular Number System. In *26th IEEE International Symposium on Computer Arithmetic*, Kyoto, Japan, Jun 2019. to appear.

[DDEM+19b] Laurent-Stéphane Didier, Fangan-Yssouf Dosso, Nadia El Mrabet, Jérémy Marrez, and Pascal Véron. Randomization of arithmetic over polynomial modular number system. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 199–206. IEEE, 2019.

[DDV19] Laurent-Stephane Didier, Fangan Yssouf Dosso, and Pascal Véron. Efficient modular operations using the adapted modular number system. Submitted to the Journal of Cryptographic Engineering of Springer. First version available at : https ://arxiv.org/abs/1901.11485, 2019.

[DDV20] Laurent-Stéphane Didier, Fangan-Yssouf Dosso, and Pascal Véron. Efficient modular operations using the adapted modular number system. *Journal of Cryptographic Engineering*, pages 1–23, 2020.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6) :644–654, 1976.

[DK91] Stephen R. Dussé and Burton S. Kaliski. A cryptographic library for the motorola dsp56000. In Ivan Bjerre Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, pages 230–244, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[DL92] Brandon Dixon and Arjen K Lenstra. Massively parallel elliptic curve factoring. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 183–193. Springer, 1992.

[Dos20] Fangan Yssouf Dosso. *Contribution de l'arithmétique des ordinateurs aux implémentations résistantes aux attaques par canaux auxiliaires*. PhD thesis, Université de Toulon, 2020.

[DZS14] Luka Daoud, Dawid Zydek, and Henry Selvaraj. A survey of high level synthesis languages, tools, and compilers for reconfi-

gurable high performance computing. In *Advances in Systems Science*, pages 483–492. Springer, 2014.

[EG12] Nadia El Mrabet and Nicolas Gama. Efficient multiplication over extension fields. In *WAIFI*, volume 7369 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2012.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4) :469–472, 1985.

[Eve90] Shimon Even. Systolic modular multiplication. In *Conference on the Theory and Application of Cryptography*, pages 620–624. Springer, 1990.

[Fei12] Tom Feist. Vivado design suite. *White Paper*, 5 :30, 2012.

[GJM10] Raveen R Goundar, Marc Joye, and Atsuko Miyaji. Co-z addition formulæ and binary ladders on elliptic curves. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 65–79. Springer, 2010.

[Gui10] Nicolas Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 48–64. Springer, 2010.

[HAHH06] Panu Hamalainen, Timo Alho, Marko Hannikainen, and Timo D Hamalainen. Design and implementation of low-area and low-power aes encryption hardware core. In *9th EUROMICRO conference on digital system design (DSD'06)*, pages 577–583. IEEE, 2006.

[HG14] Ekawat Homsirikamol and Kris Gaj. Can high-level synthesis compete against a hand-written code in the cryptographic domain ? a case study. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–8. IEEE, 2014.

[HM11] Darrel Hankerson and Alfred Menezes. *Elliptic curve cryptography*. Springer, 2011.

[jM19] jérémy Marrez. *Représentations adaptées à l'arithmétique modulaire et à la résolution de systèmes flous*. PhD thesis, SORBONNE UNIVERSITy, France, 2019.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1) :36–63, 2001.

[JTB+01] S Janssens, J Thomas, W Borremans, P Gijsels, Ingrid Verbauwhede, Fréderik Vercauteren, Bart Preneel, and Joos Vandewalle. Hardware/software co-design of an elliptic curve public-key cryptosystem. In *2001 IEEE Workshop on Signal Processing Systems. SiPS 2001. Design and Implementation (Cat. No. 01TH8578)*, pages 209–216. IEEE, 2001.

[JY02] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International workshop on cryptographic hardware and embedded systems*, pages 291–302. Springer, 2002.

[KAK96] C Kaya Koc, Tolga Acar, and Burton S Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE micro*, 16(3) :26–33, 1996.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[KKSS00] Shinichi Kawamura, Masanobu Koike, Fumihiko Sano, and Atsushi Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 523–538. Springer, 2000.

[Knu81] Donald E Knuth. The art of computer programming vol. 2 : Seminumerical methods, 1981.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177) :203–209, 1987.

[Kob89] Neal Koblitz. Hyperelliptic cryptosystems. *Journal of cryptology*, 1(3) :139–150, 1989.

[Kor94] Peter Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on computers*, 43(8) :892–898, 1994.

[KYKS16] Shin-ichi Kawamura, Tomoko Yonemura, Yuichi Komano, and Hideo Shimizu. Exact error bound of cox-rower architecture for rns arithmetic. *IACR Cryptol. ePrint Arch.*, 2016 :266, 2016.

[LHL03] Chin-Bou Liu, Chua-Huang Huang, and Chin-Laung Lei. Design and implementation of long-digit karatsuba's multiplication algorithm using tensor product formulation. In *The Ninth Workshop on Compiler Techniques for High-Performance Computing*, pages 23–30, 2003.

[LLMP93] Arjen K Lenstra, Hendrik W Lenstra, Mark S Manasse, and John M Pollard. The factorization of the ninth fermat number. *Mathematics of Computation*, 61(203) :319–349, 1993.

[LTPK19] Carson Labrado, Himanshu Thapliyal, Stacy Prowell, and Teja Kuruganti. Use of thermistor temperature sensors for cyber-physical system security. *Sensors*, 19(18) :3905, 2019.

[MAGK16] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. Adaptive threshold non-pareto elimination : Re-thinking machine learning for system level design space exploration on fpgas. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 918–923. IEEE, 2016.

[Mel07] Nicolas Meloni. New point addition formulae for ecc applications. In *International Workshop on the Arithmetic of Finite Fields*, pages 189–201. Springer, 2007.

[Mil86] Mil1_86 Victor S Miller. Use of elliptic curves in cryptography ; crypto'85, lncs 218, 1986.

[Mil97] Victor S Miller. Elliptic curves and their use in cryptography. In *DIMACS Workshop on Unusual Applications of Number Theory*, volume 21. sn, 1997.

[MML+16] Amine Mrabet, Nadia El Mrabet, Ronan Lashermes, Jean-Baptiste Rigaud, Belgacem Bouallegue, Sihem Mesnager, and Mohsen Machhout. A systolic hardware architectures of montgomery modular multiplication for public key cryptosystems. *IACR Cryptology ePrint Archive*, 2016 :487, 2016.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170) :519–521, 1985.

[MS09] Grant Martin and Gary Smith. High-level synthesis : Past, present, and future. *IEEE Design & Test of Computers*, 26(4) :18–25, 2009.

[MSFUCAB11] Miguel Morales-Sandoval, Claudia Feregrino-Uribe, Rene Cumplido, and Ignacio Algredo-Badillo. A reconfigurable gf (2 m)

elliptic curve cryptographic coprocessor. In *2011 VII Southern Conference on Programmable Logic (SPL)*, pages 209–214. IEEE, 2011.

[NP08] Christophe Negre and Thomas Plantard. Efficient modular arithmetic in adapted modular number system using lagrange representation. In *Australasian Conference on Information Security and Privacy*, pages 463–477. Springer, 2008.

[NSA08] National security agency, the case for elliptic curve cryptography, 2008.

[OBPV03] S. B. Ors, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a montgomery modular multiplier in a systolic array. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 8 pp.–, April 2003.

[OD04] Etienne Ogoubi and Jean Pierre David. Automatic synthesis from high level asm to vhdl : a case study. In *The 2nd Annual IEEE Northeast Workshop on Circuits and Systems, 2004. NEWCAS 2004.*, pages 81–84. IEEE, 2004.

[PL12] Programmable Logic PL. Zynq-7000 all programmable soc overview. 2012.

[Pla05] Thomas Plantard. *Arithmétique modulaire pour la cryptographie*. PhD thesis, Montpellier 2 University, France, 2005.

[PP09] Christof Paar and Jan Pelzl. *Understanding cryptography : a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[QR89] Patrice Quinton and Yves Robert. *Algorithmes et architectures systoliques*. Masson, 1989.

[RK14] Abdalhossein Rezai and Parviz Keshavarzi. High-throughput modular multiplication and exponentiation algorithms using multibit-scan–multibit-shift technique. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(9) :1710–1719, 2014.

[RK15] Abdalhossein Rezai and Parviz Keshavarzi. Compact sd : A new encoding algorithm and its application in multiplication. *International journal of computer mathematics*, 94(3) :554–569, 2015.

[RK16] Abdalhossein Rezai and Parviz Keshavarzi. High-performance scalable architecture for modular multiplication using a new digit-serial computation. *Microelectronics journal*, 55 :169–178, 2016.

[RSA78a] Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[RSA78b] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[ŠFD03] Martin Šimka, Viktor Fischer, and Miloš Drutarovskỳ. Hardware-software codesign in embedded asymmetric cryptography application–a case study. In *International Conference on Field Programmable Logic and Applications*, pages 1075–1078. Springer, 2003.

[Šim03] Martin Šimka. Rsa implementation on reconfigurable hardware. In *Proceedings of the III. PhD student conference*, pages 81–82. Citeseer, 2003.

[TK99] Alexandre F Tenca and Cetin K Koç. A scalable architecture for montgomery nultiplication. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 94–108. Springer, 1999.

[VSK20] Priyanshi Vishnoi, SL Shimi, and Adesh Kumar. Symmetric cryptography and hardware chip implementation on fpga. In *Intelligent Communication, Control and Devices*, pages 945–955. Springer, 2020.

[WCSG03] Xingjun Wu, Hongyi Chen, Yihe Sun, and Weixin Gai. A fully-pipeline linear systolic architecture for modular multiplier in public-key crypto-systems. *Journal of VLSI signal processing systems for signal, image and video technology*, 33(1-2) :191–197, 2003.

[Xil14a] VH Xilinx. Vivado design suite user guide-high-level synthesis, 2014.

[Xil14b] I Xilix. Zynq-7000 all programmable soc : Technical reference manual. ug585, v1. 8.1, 2014.

Résumé

CHAOUCH Asma

Laboratoire IMATH, Université de Toulon

Laboratoire $E\mu E$, Université de Monastir

Implémentation efficace de l'arithmétique AMNS

Résumé Cette thèse de doctorat s'intitule Implantation efficace de l'arithmétique sur le système de représentation modulaire adapté (AMNS : Adapted Modular Number System). Elle porte sur l'étude et la validation des architectures matérielles de l'arithmétique modulaire et de la multiplication scalaire en AMNS sur FPGA. Nous avons conçu deux architectures matérielles pour implanter la multiplication modulaire en AMNS sur le corps finis $\mathbb{F}_p$. Nous avons présenté les résultats de nos implantations pour des différentes taille de p. Les résultats montrent que nos architectures sont compétitifs par rapport à l'état de l'art. Pour la multiplication scalaire, nous abordons le système de coordonnées Co-Z proposé par Méloni en 2007. L'objectif est de concevoir une architecture matérielle de la multiplication scalaire en utilisant le système de représentation modulaire adapté.

Mots clés : AMNS, Arithmétique modulaire, Implantation matérielle, VHDL, multiplication modulaire, architectures matérielles, multiplication scalaire.

Efficient implementation of AMNS arithmetic

Abstract This thesis is entitled Efficient implementation of arithmetic on the Adapted Modular Number System (AMNS). It deals with the study and validation of hardware architectures for modular arithmetic and scalar multiplication in AMNS on FPGA. We have designed two hardware architectures to implement modular multiplication in AMNS on the finite field $\mathbf{F}_p$. We have presented the results of our implementations for different sizes of p. The results show that our architectures are competitive with the state of the art. For scalar multiplication, we address the Co-Z coordinate system proposed by Méloni in 2007. The objective is to design a hardware architecture for scalar multiplication using the appropriate modular representation system.

Keywords : AMNS, Modular arithmetic, Hardware implementation, VHDL, modular multiplication, hardware architectures, scalar multiplication.